

MANUALE DI

JAVA 8

Claudio De Sio Cesari

**MANUALE DI
JAVA 8**

**Programmazione orientata agli oggetti
con Java Standard Edition 8**



EDITORE ULRICO HOEPLI MILANO

Copyright © Ulrico Hoepli Editore S.p.A. 2014

via Hoepli 5, 20121 Milano (Italy)

tel. +39 02 864871 – fax +39 02 8052886

e-mail hoepli@hoepli.it

Seguici su Twitter: [@Hoepli_1870](#)

www.hoepli.it

Tutti i diritti sono riservati a norma di legge
e a norma delle convenzioni internazionali

ISBN: 978-88-203-6400-7

Progetto editoriale: Emanuele Giuliani (emanuele@giuliani.mi.it)

Copertina: Sara Taglialegne

Realizzazione digitale: Promedia, Torino

Prefazione

Parte I - Le basi del linguaggio

Modulo 1 - Introduzione a Java 8

1.1 Introduzione a Java

1.1.1 Cos'è Java

1.2 Ambiente di sviluppo

1.2.1 Ambienti di sviluppo più complessi

1.3 Struttura del JDK

1.3.1 Guida dello sviluppatore passo dopo passo

1.4 Primo approccio al codice

1.5 Analisi del programma “HelloWorld”

1.6 Compilazione ed esecuzione del programma HelloWorld

1.7 Possibili problemi in fase di compilazione ed esecuzione

1.7.1 Possibili messaggi di errore in fase di compilazione

1.7.2 Possibili messaggi relativi alla fase di interpretazione

Riepilogo

Esercizi modulo 1

Soluzioni esercizi modulo 1

Modulo 2 - Componenti fondamentali di un programma Java

2.1 Componenti fondamentali di un programma Java

2.1.1 Convenzione per la programmazione Java

2.2 Le basi della programmazione object oriented: classi ed oggetti

2.2.1 Osservazione importante sulla classe Punto

2.2.2 Osservazione importante sulla classe Principale

2.2.3 Un'altra osservazione importante

2.3 I metodi in Java

2.3.1 Dichiarazione di un metodo

2.3.2 Chiamata (o invocazione) di un metodo

2.3.3 Varargs

2.4 Le variabili in Java

2.4.1 Dichiarazione di una variabile

2.4.2 Variabili d'istanza

2.4.3 Variabili locali

2.4.4 Parametri formali

2.5 I metodi costruttori

2.5.1 Caratteristiche di un costruttore

2.5.2 Costruttore di default

2.6 Package

Riepilogo

Esercizi modulo 2

Soluzioni esercizi modulo 2

Modulo 3 - Identificatori, tipi di dati e array

3.1 Stile di codifica

3.1.1 Schema libero

3.1.2 Case sensitive

3.1.3 Commenti

3.1.4 Regole per gli identificatori

3.1.5 Standard e convenzioni per i nomi

3.2 Tipi di dati primitivi

3.2.1 Tipi di dati interi, casting e promotion

3.2.2 Tipi di dati a virgola mobile, casting e promotion

3.2.3 Underscore in tipi di dati numerici

3.2.4 Tipo di dato logico - booleano

3.2.5 Tipo di dato primitivo letterale

3.3 Tipi di dati non primitivi: reference

3.3.1 Passaggio di parametri per valore

3.3.2 Inizializzazione delle variabili d'istanza

3.4 Introduzione alla libreria standard

3.4.1 Il comando `import`

3.4.2 La classe `String`

3.4.3 La documentazione della libreria standard di Java

3.4.4 Lo strumento `javadoc`

3.5 Gli array in Java

3.5.1 Dichiarazione

3.5.2 Creazione

3.5.3 Inizializzazione

3.5.4 Array multidimensionali

3.5.5 `String args[]`

Riepilogo

Esercizi modulo 3

Soluzioni esercizi modulo 3

Modulo 4 - Operatori e gestione del flusso di esecuzione

4.1 Operatori di base

- 4.1.1 Operatore d'assegnazione
- 4.1.2 Operatori aritmetici
- 4.1.3 Operatori (unari) di pre e post-incremento (e decremento)
- 4.1.4 Operatori bitwise
- 4.1.5 Operatori relazionali o di confronto
- 4.1.6 Operatori logico – booleani
- 4.1.7 Concatenazione di stringhe con +
- 4.1.8 Priorità degli operatori

4.2 Gestione del flusso di esecuzione

4.3 Costrutti di programmazione semplici

- 4.3.1 Il costrutto if
- 4.3.2 L'operatore ternario
- 4.3.3 Il costrutto while

4.4 Costrutti di programmazione avanzati

- 4.4.1 Il costrutto for
- 4.4.2 Il costrutto do
- 4.4.3 Ciclo for migliorato
- 4.4.4 Il costrutto switch
- 4.4.5 Due importanti parole chiave: break e continue

Riepilogo

Esercizi modulo 4

Soluzioni esercizi modulo 4

Parte II - Object Orientation

Modulo 5 - Incapsulamento e visibilità

5.1 Breve storia della programmazione ad oggetti

5.2 I paradigmi della programmazione ad oggetti

- 5.2.1 Astrazione e riuso

5.3 Incapsulamento

- 5.3.1 Prima osservazione sull'incapsulamento
- 5.3.2 Seconda osservazione sull'incapsulamento
- 5.3.3 Il reference this
- 5.3.4 Uso di this con variabili
- 5.3.5 Uso di this con metodi e costruttori

5.4 Quando utilizzare l'incapsulamento

5.5 Come usare l'incapsulamento

5.6 Gestione dei package

5.7 Gestione “a mano”

5.8 Modificatori d’accesso

5.9 Il modificatore `static`

5.9.1 Metodi statici

5.9.2 Variabili statiche (di classe)

5.9.3 Inizializzatori statici ed inizializzatori d’istanza

5.9.4 `import` statici

5.9.5 Quando usare `static`

5.9.6 Design Pattern Singleton

Riepilogo

Esercizi modulo 5

Soluzioni esercizi modulo 5

Modulo 6 - Ereditarietà e interfacce

6.1 Ereditarietà

6.1.1 La parola chiave `extends`

6.2 Il modificatore `final`

6.3 La classe `Object`

6.4 Rapporto ereditarietà-incapsulamento

6.5 Quando utilizzare l’ereditarietà

6.5.1 La relazione “is a”

6.5.2 Ereditarietà e costruttori

6.5.3 La parola chiave `super`

6.5.4 Ereditarietà e inizializzatori

6.5.5 Generalizzazione e specializzazione

6.6 Il modificatore `abstract`

6.6.1 Metodi astratti

6.6.2 Classi astratte

6.7 Interfacce

6.7.1 Definizione classica (pre-Java 8)

6.7.2 Metodi statici (Java 8)

6.7.3 Metodi di default e interfacce funzionali (Java 8)

6.7.4 Ereditarietà multipla

6.7.5 Differenze tra interfacce e classi astratte

Riepilogo

Esercizi modulo 6

Soluzioni esercizi modulo 6

Modulo 7 - Polimorfismo

7.1 Polimorfismo

7.1.1 Convenzione per i reference

7.2 Polimorfismo per metodi

7.2.1 Overload

7.2.2 Varargs

7.2.3 Override

7.2.4 Annotazione sull'override

7.3 Polimorfismo per dati

7.3.1 Parametri polimorfi

7.3.2 Collezioni eterogenee

7.3.3 Casting di oggetti

7.3.4 Invocazione virtuale dei metodi

7.3.5 Esempio d'utilizzo del polimorfismo

7.3.6 Polimorfismo e interfacce

Riepilogo

Esercizi modulo 7

Soluzioni esercizi modulo 7

Modulo 8 - Eccezioni e asserzioni

8.1 Eccezioni, errori e asserzioni

8.2 Gerarchie e categorizzazioni

8.3 Meccanismo per la gestione delle eccezioni

8.4 Try with resources

8.5 Eccezioni personalizzate e propagazione dell'eccezione

8.5.1 Warnings

8.5.2 Precisazione sull'override

8.6 Introduzione alle asserzioni

8.6.1 Progettazione per contratto

8.6.2 Note per l'esecuzione di programmi che utilizzano la parola `assert`

8.6.3 Quando usare le asserzioni

 Consiglio 1)

 Consiglio 2)

 Consiglio 3)

8.6.4 Conclusioni

Riepilogo

Esercizi modulo 8

Soluzioni esercizi modulo 8

Parte III - Caratteristiche avanzate

Modulo 9 - Enumerazioni e tipi innestati

9.1 Classi innestate: classi interne

9.1.1 Classe innestata: definizione

9.1.2 Classi innestate: proprietà

9.1.3 Quando usare le classi innestate

9.2 Classi anonime: definizione

9.2.1 Quando usare le classi anonime

9.3 Tipi Enumerazioni

9.3.1 Ereditarietà e polimorfismo con enum

9.3.2 Metodi, variabili, costruttori e tipi innestati in un'enumerazione

9.3.3 Quando utilizzare un'enum

9.3.4 Enumerazioni innestate (in classi) o enumerazioni membro

9.3.5 Enumerazioni e metodi specifici degli elementi

9.3.6 Switch e enum

Riepilogo

Esercizi modulo 9

Soluzioni esercizi modulo 9

Modulo 10 - Tipi Generici

10.1 Generics e tipi parametro

10.1.1 Generics e collection

10.1.2 Dietro le quinte

10.1.3 Tipi primitivi e autoboxing-autounboxing

10.1.4 Interfaccia Iterator

10.1.5 Interfaccia Map

10.2 Ereditarietà dei Generics e Type erasure

10.2.1 Wildcard

10.3 Creare propri tipi generici

10.3.1 Parametri e wildcard bounded

10.3.2 Metodi generici

10.4 Deduzione automatica del tipo

10.4.1 Compilazione

10.4.2 Parametri covarianti

10.4.3 Cast automatico di reference al loro tipo “intersezione” nelle operazioni condizionali

10.4.4 Wildcard capture e metodi helper

Riepilogo

Esercizi modulo 10

Soluzioni esercizi modulo 10

Modulo 11 - La libreria indispensabile: il package java.lang

11.1 Introduzione al package `java.lang`

- 11.1.1 La classe `String`
- 11.1.2 La classe `Object`: metodi `toString()`, `clone()`, `equals()` e `hashCode()`
- 11.1.3 La classe `System`
- 11.1.4 La classe `Runtime`
- 11.1.5 La classe `Class` e `Reflection`
- 11.1.6 Le classi wrapper
- 11.1.7 Autoboxing-Autounboxing
 - 11.1.7.1 Assegnazione di un valore `null` al tipo wrapper
 - 11.1.7.2 Costrutti del linguaggio ed operatori relazionali
 - 11.1.7.3 Overload
- 11.1.8 La classe `Math`
- 11.1.9 Ordinamento: le interfacce `Comparable` e `Comparator`

Riepilogo

Esercizi modulo 11

Soluzioni esercizi modulo 11

Modulo 12 - Tipi annotazioni

12.1 Definizione di annotazione (metadato)

- 12.1.1 Primo esempio
- 12.1.2 Tipologie di annotazioni e sintassi
 - 12.1.2.1 Annotazione ordinaria (o completa)
 - 12.1.2.2 Annotazione a valore unico
 - 12.1.2.3 Annotazione segnalibro

12.2 Annotare annotazioni (meta-annotazioni)

- 12.2.1 `Target`
 - 12.2.1.1 Type Annotations
- 12.2.2 `Retention`
- 12.2.3 `Documented`
- 12.2.4 `Inherited`
- 12.2.5 `Repeatable`

12.3 Annotazioni standard

- 12.3.1 `Override`
- 12.3.2 `Deprecated`
- 12.3.3 `SuppressWarnings`
- 12.3.4 `FunctionalInterface`
- 12.3.5 `Native`

Riepilogo

Esercizi modulo 12

Soluzioni esercizi modulo 12

Modulo 13 - Le librerie di utilità: il package `java.util` e Date-Time API

13.1 Package `java.util`

- 13.1.1 Le classi `Properties` e `Preferences`
- 13.1.2 Classe `Locale` ed internazionalizzazione
- 13.1.3 La classe `ResourceBundle`
- 13.1.4 La classe `StringTokenizer`
- 13.1.5 Formattazioni di output
- 13.1.6 Espressioni regolari
- 13.1.7 Date, orari e valute

13.2 La novità di Java 8: Date-Time API

- 13.2.1 Standardizzazione dei metodi
- 13.2.2 Il package `java.time`
 - 13.2.2.1 La classe `Instant`
 - 13.2.2.2 Le classi `Duration` e `Period`
 - 13.2.2.3 La enumerazioni `DayOfWeek` e `Month`
 - 13.2.2.4 Date classes: `LocalDate`, `YearMonth`, `MonthYear` e `Year`
 - 13.2.2.5 Le classi `LocalTime` e `LocalDateTime`
 - 13.2.2.6 Geolocalizzazione: le classi `ZonedDateTime`, `ZoneId` e `ZoneOffset`
- 13.2.3 Il package `java.time.format`
- 13.2.4 Il package `java.time.temporal`
 - 13.2.4.1 L'enumerazione `ChronoUnit`
 - 13.2.4.2 Temporal Adjusters
 - 13.2.4.3 Temporal Queries
- 13.2.5 Codice legacy

Riepilogo

Esercizi modulo 13

Soluzioni esercizi modulo 13

Modulo 14 - Gestione dei thread

14.1 Introduzione ai thread

- 14.1.1 Definizione provvisoria di thread
- 14.1.2 Cosa significa multithreading

14.2 La classe Thread e la dimensione temporale

- 14.2.1 Analisi di `ThreadExists`
- 14.2.2 L'interfaccia `Runnable` e la creazione dei thread
- 14.2.3 Analisi di `ThreadCreation`
- 14.2.4 La classe `Thread` e la creazione dei thread

14.3 Priorità, scheduler e sistemi operativi

- 14.3.1 Analisi di `ThreadRace`
- 14.3.2 Comportamento Windows (time slicing o round-robin scheduling)
- 14.3.3 Comportamento Unix (preemptive scheduling)
- 14.3.4 Il modificatore `volatile`

14.4 Thread e sincronizzazione

14.4.1 Analisi di `Synch`

14.4.2 Monitor e Lock

14.5 La comunicazione fra thread

14.5.1 Analisi di `IdealEconomy`

14.6 Concorrenza

14.6.1 Oggetti Immutabili

14.6.2 Tecnologie Java e classi standard ausiliarie

14.6.3 Libreria Java sulla concorrenza

 14.6.3.1 Il package `java.util.concurrent.locks`

 14.6.3.2 Il package `java.util.concurrent.atomic`

 14.6.3.3 Interfacce Executors

 14.6.3.4 La classe `Semaphore`

 14.6.3.5 La classe `CyclicBarrier`

Riepilogo

Esercizi modulo 14

Soluzioni esercizi modulo 14

Modulo 15 - Espressioni Lambda

15.1 Espressioni lambda

15.1.1 Sintassi

15.1.2 Comprendere le espressioni lambda

15.1.3 Espressione Lambda vs Classe Anonima

 15.1.3.1 Sinteticità

 15.1.3.2 Regole e visibilità

 15.1.3.3 Dinamicità

 15.1.3.4 Gestione delle eccezioni

 15.1.3.5 Quando usare le espressioni lambda

15.2 Reference a metodi

15.2.1 Sintassi

15.2.2 Reference a un metodo statico

15.2.3 Reference a un metodo d'istanza

15.2.4 Reference a un metodo d'istanza di un certo tipo

15.2.5 Reference a un costruttore

15.3 Le interfacce funzionali del package `java.util.function`

15.3.1 `Predicate`

15.3.2 `Consumer`

15.3.3 `Supplier`

15.3.4 `Function`

15.3.5 `UnaryOperator` e composizione di espressioni lambda

Riepilogo

Esercizi modulo 15

Soluzioni esercizi modulo 15

Modulo 16 - Collections Framework e Stream API

16.1 Introduzione al framework Collections

16.2 L'interfaccia Collection

16.2.1 Iterare sulle collezioni

16.2.1.1 Ciclo foreach

16.2.1.2 Iteratori

16.2.1.3 Metodo `forEach()` dell'interfaccia `Iterable`

16.3 Interfaccia List

16.3.1 Implementazioni di `List`

16.4 Le interfacce Set e SortedSet

16.4.1 Implementazioni di `Set` e `SortedSet`

16.5 Le interfacce Queue e Deque

16.5.1 L'interfaccia `BlockingQueue` e implementazioni di `Queue` e `Deque`

16.6 Le interfacce Map e SortedMap

16.6.1 Implementazioni di `Map` e `SortedMap`

16.7 Algoritmi e utilità

16.8 Introduzione alla libreria Stream API

16.8.1 Definizioni di `Stream` e pipeline

16.8.2 Classi `Optional`

16.8.3 Reduction Operations

16.8.3.1 Metodo `reduce()`

16.8.3.2 Metodo `collect()` e la classe `Collectors`

16.8.4 Stream paralleli e prestazioni

Riepilogo

Esercizi modulo 16

Soluzioni esercizi modulo 16

Modulo 17 - Input-Output

17.1 Introduzione all'input-output

17.2 Pattern Decorator

17.2.1 Descrizione del pattern

17.3 Descrizione del package

17.3.1 I Character Stream

17.3.2 I Byte Stream

17.3.3 Le superinterfacce principali

17.3.4 Chiusura degli stream

17.4 Input ed output "classici"

- 17.4.1 Lettura di input da tastiera
- 17.4.2 Gestione dei file
- 17.4.3 Serializzazione di oggetti

17.5 NIO 2.0

- 17.5.1 L'interfaccia Path
- 17.5.2 La classe Files

Riepilogo

Esercizi modulo 17

Soluzioni esercizi modulo 17

Modulo 18 - Java Database Connectivity

18.1 Introduzione a JDBC

18.2 Le basi di JDBC

- 18.2.1 Implementazione del vendor (Driver JDBC)
- 18.2.2 Implementazione dello sviluppatore (Applicazione JDBC)
- 18.2.3 Analisi dell'esempio JDBCApp

18.3 Altre caratteristiche di JDBC

- 18.3.1 Indipendenza dal database
- 18.3.2 Operazioni CRUD
- 18.3.3 Statement parametrizzati
- 18.3.4 Stored procedure
- 18.3.5 Mappatura dei tipi Java - SQL
- 18.3.6 Transazioni

18.4 Evoluzione di JDBC

- 18.4.1 JDBC 2.0
- 18.4.2 JDBC 3.0
- 18.4.3 JDBC 4.0, 4.1 e 4.2

Riepilogo

Esercizi modulo 18

Soluzioni esercizi modulo 18

Modulo 19 - Interfacce grafiche: introduzione a JavaFX

19.1 Storia delle GUI in Java

19.2 Caratteristiche di JavaFX

19.3 Hello World

- 19.3.1 Analisi di `HelloWorld`
- 19.3.2 Esecuzione di un'applicazione JavaFX

19.4 Creazione di interfacce complesse con i Layout

- 19.4.1 I Layout pane di JavaFX
 - 19.4.1.1 Le classi `VBox` e `HBox`

- 19.4.1.2 La classe `BorderPane`
- 19.4.1.3 La classe `GridPane`
- 19.4.1.4 Altri Layout Pane
- 19.4.1.5 FXML

19.5 CSS

19.6 Gestione degli eventi

19.7 Proprietà JavaFX e Bindings

- 19.7.1 Proprietà JavaFX

19.8 Effetti speciali

19.9 Componenti grafici avanzati

Riepilogo

Esercizi modulo 19

Soluzioni esercizi modulo 19

Indice analitico

Informazioni sul Libro

Circa l'autore

Manuale di Java 8 è strutturato in modo tale da facilitare l'apprendimento del linguaggio, anche a chi non ha mai programmato, o a chi ha programmato con linguaggi funzionali. Infatti, la forma espositiva, la struttura e i contenuti del testo, sono stati accuratamente scelti basandosi sull'esperienza che ho accumulato come formatore e mentore. In particolare, per Sun Educational Services Italia (ora Oracle) creatrice di Java, ho avuto la possibilità di erogare corsi per migliaia di discenti su tecnologia Java, analisi e progettazione object oriented ed UML.

Questo testo inoltre, è stato creato sulle ceneri di altri due manuali su Java di natura free: "Il linguaggio Object Oriented Java" e "Object Oriented && Java 5". Entrambi questi manuali sono stati scaricati decine di migliaia di volte da <http://www.claudiodesio.com> e da molti importanti mirror specializzati nel settore, riscuotendo tantissimi consensi da parte dei lettori. In particolare "Object Oriented && Java 5" è stato scaricato oltre 350000 volte dai programmati italiani.

Inoltre anche le precedenti versioni cartacee di questo libro (*Manuale di Java 6* e *Manuale di Java 7* sempre editi da Hoepli) hanno avuto grande successo vendendo migliaia di copie, e sono stati usati come testo di riferimento in molte facoltà universitarie italiane per corsi relativi alla programmazione.

Il libro che avete tra le mani è quindi il risultato di un lavoro di anni, basato non soltanto sulla sola esperienza didattica, ma anche su centinaia di feedback di lettori.

Cosa c'è di nuovo

La versione 8 (o 1.8) di Java può essere considerata la più rivoluzionaria della storia del linguaggio! La sintassi infatti si è arricchita di nuovi costrutti che rendono il linguaggio più potente e compatto. Persino la naturale propensione alla programmazione Object Oriented è ora stata sconvolta e potenziata dall'introduzione di potenti strumenti tipici della programmazione funzionale moderna. Java 8 è un linguaggio nuovo, molto diverso dal linguaggio che abbiamo usato sino ad oggi. Le potenzialità sono aumentate ancora, e si candida a diventare sempre di più il leader dei linguaggi di programmazione anche nei prossimi anni. L'introduzione delle Stream API, delle espressioni Lambda, del nuovo supporto al multithreading, della nuova Date & Time API, dell'implementazione dei metodi di default nelle interfacce, di JavaFX come nuovo standard per le GUI, del nuovo modo di gestire le collezioni e dell'intendere l'ereditarietà multipla, obbligano chiunque ad aggiornarsi.

Struttura del testo

Manuale di Java 8 è diviso in tre parti. Nella prima parte denominata "Le basi di Java", verranno presentati i concetti che permetteranno anche a chi inizia da zero, di cominciare a programmare. Nella seconda parte "Object Orientation" verranno spiegati i concetti fondamentali per creare correttamente programmi partendo da zero. Nella terza parte "Caratteristiche avanzate" saranno introdotti (e approfonditi) tutti gli argomenti più complessi. Si tratta quindi di un testo che dovrebbe soddisfare le aspettative sia dell'aspirante programmatore che del programmatore esperto. Infatti

nella prima parte (composta da soli quattro moduli) nulla verrà dato per scontato, e il lettore sarà supportato con esempi, esercizi e spiegazioni semplificate. Ho anche creato un semplice tool di sviluppo gratuito e open source di nome EJE, per supportare le fasi di apprendimento iniziali. Lo studio dei primi quattro moduli, dovrebbe quindi rendere il lettore già in grado di scrivere i primi programmi ed avere una certa confidenza con l'ambiente di sviluppo.

Nei moduli della seconda parte (“Object Orientation”) sono trattati alcuni argomenti che difficilmente si trovano su altri testi similari. I moduli che vanno dal cinque all’otto, pongono particolare enfasi sul supporto che il linguaggio offre all’Object Orientation. Solitamente infatti, la difficoltà maggiore che incontra un programmatore Java è nel riuscire a mettere in pratica proprio i paradigmi della programmazione ad oggetti. Questo testo si sforza quindi di fornire tutte le informazioni necessarie al lettore, per intraprendere la strada della programmazione Java, nel modo più corretto possibile, ovvero in maniera object oriented.

La terza parte (“Caratteristiche Avanzate”) che comprende i moduli dal nono al diciannovesimo, tratta tutte le caratteristiche avanzate del linguaggio, e il grado di approfondimento è particolarmente elevato. Si imparerà ad affiancare alla programmazione object oriented altri tipi di programmazione come quella generica, quella basata sui metadati, quella basata sui contratti, quella multithreaded e quella funzionale.

Inoltre, per non gravare troppo sulla corposità dell’opera, moltissimo materiale (svariate appendici e centinaia di esercizi supplementari) è stato spostato in un apposito spazio disponibile online agli indirizzi <http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

A chi si rivolge

Manuale di Java 8 è stato strutturato per soddisfare le aspettative di:

- Aspiranti programmatori che vogliono entrare nel mondo del lavoro: nulla è dato per scontato, è possibile imparare a programmare anche partendo da zero, ed entrare nel mondo del lavoro dalla porta principale.
- Studenti universitari: le precedenti versioni di questa opera sono state adottate come libro di testo per molti corsi universitari nelle maggiori università italiane.
- Programmatori Java esperti che devono aggiornarsi alla versione 8: ricordiamo si tratta della versione più rivoluzionaria mai realizzata nella quasi ventennale storia di Java.

Autore

Dal 1999 lavoro come consulente IT free lance. Oggi sono specialista in formazione, technical writing, sviluppo, analisi, progettazione, tecnologie Java, architettura e metodologie object oriented. Sono autore di diversi articoli tecnici, e dei libri *Manuale di Java 6* e *Manuale di Java 7* entrambi editi con Hoepli. Ho collaborato con diverse Università, Enti Ministeriali e aziende tra cui Sun Microsystems (ora Oracle) creatrice di Java, come trainer e mentor. È possibile contattarmi tramite i seguenti canali:

- E-mail: claudio@claudiodesio.com
- Facebook: <http://www.facebook.com/claudiodesiocesari>
- Twitter: <http://twitter.com/cdesio>
- LinkedIn: <http://www.linkedin.com/in/claudiodesio>
- Google+: <http://google.com/+ClaudioDeSioCesari>



Materiali online

Sul sito web di Hoepli <http://www.hoeplieditore.it/6291-1> e sul sito dell'autore Claudio De Sio <http://www.claudiodesio.com/java8.html> si trovano anche le numerose appendici a completamento dell'opera che l'arricchiscono di ulteriore valore.

Codice sorgente

Lavorando con gli esempi di codice presenti in questo libro, si può decidere di scrivere tutto il codice a mano oppure di usare i file sorgenti che accompagnano il libro. Tutto il codice sorgente utilizzato in questo libro è disponibile per il download agli indirizzi <http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Errata

È stato compiuto ogni sforzo possibile per evitare errori nel testo e nel codice. Tuttavia, nessuno è perfetto, e gli sbagli capitano. Se si dovesse trovare un errore in uno dei libri Hoepli, come un errore ortografico o una parte di codice non funzionante, vi preghiamo di segnalarlo per e-mail a hoepli@hoepli.it. La segnalazione verrà messa in errata corrigere; altri lettori eviteranno ore di frustrazione e allo stesso tempo si contribuisce a garantire un livello di qualità dell'informazione sempre maggiore. Per trovare l'errata sul web, si faccia riferimento agli indirizzi <http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Ringraziamenti

Il mio ringraziamento è per tutte le persone che mi hanno sostenuto in questo lavoro, e a tutti coloro che in qualche modo hanno contribuito alla sua realizzazione (supportandomi o sopportandomi!).

Un grazie particolare va a Emanuele Giuliani che ha curato quest'opera con grande pazienza e

professionalità.

Claudio De Sio Cesari

A Rosalia, Andrea e Simone...

Parte I

Le basi del linguaggio

La parte I è pensata per permettere un approccio a Java “non traumatico” a chiunque. Lo studio di questa sezione infatti, dovrebbe risultare utile sia al neofita che vuole iniziare a programmare da zero, sia al programmatore esperto che ha bisogno solo di rapide consultazioni. Infatti è presente un’esaurente copertura dei concetti fondamentali del linguaggio quali classi, oggetti, metodi, operatori, costruttori e costrutti di controllo. Inoltre vengono riportati in dettaglio tutti i passi che servono per creare applicazioni Java, senza dare per scontato niente. Vengono quindi introdotti la storia, le caratteristiche, l’ambiente di sviluppo e viene spiegato come consultare la documentazione ufficiale. Al termine dello studio di questa parte, il lettore dovrebbe possedere tutte le carte in regola per iniziare a programmare in Java.

Introduzione a Java 8

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Saper definire il linguaggio di programmazione Java e le sue caratteristiche (unità 1.1).
- ✓ Interagire con l'ambiente di sviluppo: il Java Development Kit (unità 1.2, 1.3, 1.6, 1.7).
- ✓ Saper digitare, compilare e mandare in esecuzione una semplice applicazione (unità 1.4, 1.5).

L'obiettivo primario di questo modulo è quello di permettere al lettore di ottenere subito dei primi risultati concreti. Al termine del modulo avremo creato, compilato ed eseguito un primo programma Java. Si tratta di un programma molto semplice, ma questo ci permetterà di prendere subito confidenza con la programmazione Java. Per raggiungere lo scopo non daremo nulla per scontato: definiremo il linguaggio e la tecnologia Java, descriveremo l'ambiente di sviluppo, mostreremo un processo che passo dopo passo ci permetterà di eseguire la nostra prima applicazione, impareremo a comprendere gli eventuali messaggi di errore e a risolvere i primi problemi di programmazione. In questo modo anche il lettore che non ha basi di programmazione potrà raggiungere lo scopo.

È del tutto probabile che senza basi di programmazione si incontrino delle difficoltà nel capire alcuni concetti, ma non bisogna scoraggiarsi. In questo primo modulo c'è tutto il necessario per iniziare da zero, ma bisogna essere sempre concentrati e non è consigliabile saltare frasi o addirittura paragrafi. Se qualcosa non risulta essere chiaro, è consigliata una rilettura.

1.1 Introduzione a Java

Non è possibile iniziare senza una breve introduzione su Java. A differenza delle precedenti edizioni di questo libro, non ci dilungheremo narrandovi delle eroiche gesta dei creatori di Java, e di tutte le leggende e le previsioni per il futuro. Certo, se avete comprato questo libro vi aspetterete che soddisfi tutte le vostre curiosità, ma sono davvero troppe! Su precisa richiesta dell'editore abbiamo deciso che diversi argomenti non troveranno posto tra i paragrafi di questo libro, ma saranno spostate su apposite appendici on line. È possibile scaricarle gratuitamente su <http://www.hoepli.it>, nella pagina dedicata a questo libro. Sul sito <http://www.claudiodesio.com>, troverete indicato chiaramente il link preciso. Tra queste appendici troverete anche quella relativa alla descrizione e alla storia di Java: l'appendice A. Se vi interessa, e se siete curiosi, il momento per leggere l'appendice A è adesso! In queste pagine invece saremo estremamente sintetici e cercheremo di puntare diritto al nostro obiettivo.

1.1.1 Cos'è Java

Con il termine “Java”, solitamente ci si riferisce:

1. Al linguaggio di programmazione più usato del pianeta.
2. Ad una tecnologia che include diverse “sotto-tecnologie” che si sono affermate in diversi ambiti di utilizzo del software. Oggi giorno la tecnologia Java è la più utilizzata su applicazioni enterprise, e in tutto il mondo esistono miliardi di congegni elettronici che utilizzano tecnologia Java: SCADA, telefoni cellulari, Smartphone, decoder, smart card, robot che passeggiando su Marte etc.

In questo libro parleremo del linguaggio di programmazione e non delle tecnologie.

Fu sviluppato nel 1995 da Sun Microsystems, storica e gloriosa società americana che dal 2010 è stata assorbita da Oracle. Oggi quindi, Java è a tutti gli effetti un prodotto Oracle.

Le caratteristiche di Java principali sono:

- ❑ **Sintassi:** è piuttosto semplice e simile a quella del C e del C++. Anche il linguaggio di Microsoft C# (nato nel 2002 per contrastare Java), ha a sua volta una sintassi “estremamente simile”.
- ❑ **Gratuito:** per scrivere applicazioni commerciali non bisogna pagare licenze a nessuno. Il codice Java infatti si può scrivere anche utilizzando un qualsiasi editor di testo come il blocco note, e non per forza un complicato IDE con una costosa licenza. Esistono anche tanti strumenti di sviluppo a pagamento, che possono accelerare lo sviluppo delle applicazioni Java. Ma esistono anche eccellenti prodotti gratuiti ed open source come Eclipse e Netbeans.
- ❑ **Robustezza:** questa è soprattutto una conseguenza di una gestione delle eccezioni chiara e funzionale, e di un meccanismo automatico della gestione della memoria (Garbage Collection) che esonera il programmatore dall’obbligo di dover deallocare memoria quando ce n’è bisogno (punto tra i più delicati nella programmazione). Inoltre il compilatore Java, è molto “severo”. Il programmatore è infatti costretto a risolvere tutte le situazioni “poco chiare”, garantendo al programma maggiori chance di corretto funzionamento. La logica è: “è molto meglio ottenere un errore in compilazione che in esecuzione”.
- ❑ **Libreria e standardizzazione:** Java possiede un’enorme libreria di classi standard, ottimamente documentate. Ciò rende Java un linguaggio di alto livello, e permette anche ai neofiti di creare applicazioni complesse in breve tempo. Per esempio, è relativamente semplice gestire finestre di sistema (interfacce grafiche), collegamenti a database e connessioni di rete. E questo indipendentemente dalla piattaforma su cui si sviluppa. Inoltre, grazie alle specifiche di Oracle, non esisteranno per lo sviluppatore problemi di standardizzazione come compilatori che compilano in modo differente.
- ❑ **Indipendenza dall’architettura:** grazie al concetto di macchina virtuale, ogni applicazione, una volta compilata, potrà essere eseguita su di una qualsiasi piattaforma (per esempio un PC con sistema operativo Windows o una workstation Unix). Questa è sicuramente la caratteristica più importante di Java. Infatti, nel caso in cui si debba implementare un programma destinato a diverse piattaforme per esempio, non ci sarà la necessità di doverlo convertire radicalmente da

piattaforma a piattaforma.

- **Java Virtual Machine:** Ciò che rende di fatto possibile l'indipendenza dalla piattaforma, è la Java Virtual Machine (da ora in poi JVM), un software che svolge un ruolo da interprete (ma non solo) per le applicazioni Java. Più precisamente: dopo aver scritto il nostro programma Java, bisogna prima compilarlo (per i dettagli riguardante l'ambiente e il processo di sviluppo, vi rimandiamo al paragrafo successivo). Otterremo così, non direttamente un file eseguibile (ovvero la traduzione in linguaggio macchina del file sorgente che abbiamo scritto in Java), ma un file che contiene la traduzione del nostro listato in un linguaggio molto vicino al linguaggio macchina detto “byte code”. Una volta ottenuto questo file la JVM interpreterà il bytecode ed il nostro programma andrà finalmente in esecuzione. Quindi, se una piattaforma qualsiasi possiede una Java Virtual Machine, ciò sarà sufficiente per renderla esecutrice di bytecode. È il caso del sistema operativo mobile più diffuso (Android di Google), dove le applicazioni sono scritte in Java. Ecco perché esistono così tanti modelli diversi di Smartphone e Tablet (ma anche di fornelli, lavatrici, frigoriferi e robot...) che usano Android! Si parla di “macchina virtuale” perché questo software è stato implementato per simulare un hardware. Si potrebbe affermare che “il linguaggio macchina sta ad un computer come il bytecode sta ad una Java Virtual Machine”. Oltre che permettere l'indipendenza dalla piattaforma, la JVM garantisce a Java tanti altri vantaggi. Per esempio di essere un linguaggio multi-threaded (caratteristica di solito dei sistemi operativi), ovvero capace di mandare in esecuzione più processi in maniera parallela. Inoltre garantisce dei meccanismi di sicurezza molto potenti, la supervisione del codice da parte del Garbage Collector, validi aiuti per gestire codice al runtime e tanto altro.
- **Orientato agli oggetti:** Java ci fornisce infatti alcuni strumenti che praticamente ci obbligano a programmare ad oggetti. I paradigmi fondamentali della programmazione ad oggetti (ereditarietà, encapsulamento, polimorfismo) sono più facilmente apprezzabili e comprensibili. Java è più chiaro e schematico che qualsiasi altro linguaggio orientato agli oggetti. Sicuramente chi impara Java, potrà in un secondo momento accedere in modo più naturale alla conoscenza di altri linguaggi orientati agli oggetti, giacché avrà di certo una mentalità più orientata agli oggetti. Tuttavia con il passare del tempo il linguaggio si sta evolvendo, e si sta aprendo anche ad altri paradigmi di programmazione.
- **Facilità di sviluppo:** Java ha delle caratteristiche di lusso per gli sviluppatori. Apprezzeremo sicuramente le semplificazioni per sviluppare che ci offre Java durante la lettura di questo manuale. Abbiamo per esempio già accennato al fatto che non esiste l'aritmetica dei puntatori grazie all'implementazione della Garbage Collection. Una volta Java era pubblicizzato come un linguaggio semplice. In realtà è un linguaggio molto complesso considerandone la potenza e tenendo presente che ci obbliga ad imparare la programmazione ad oggetti. In compenso si possono ottenere risultati insperati in un tempo relativamente breve. Dalla versione 5 in poi, Java è stato pubblicizzato utilizzando al posto del termine “Simplicity”, “ease of development” (in italiano “facilità di sviluppo”). Infatti, con l'introduzione delle rivoluzionarie caratteristiche come i Generics o le Annotazioni nella versione 5, sembra aver cambiato strada. Il linguaggio è diventato più difficile da imparare, ma i programmi dovrebbero però essere più semplici da scrivere. Le novità di Java 8 poi l'hanno reso estremamente più complesso, ma incredibilmente più comodo! Insomma, imparare Java non sarà una passeggiata, ma una volta comprese le

logiche fondamentali avremo tra le mani uno strumento potentissimo.

- **Aperto:** utilizzare Java non significa muoversi in un ambiente chiuso dove tutto è standard. Dalla scelta del tool di sviluppo (Eclipse, Netbeans, etc.) all'interazione con altri linguaggi, librerie esterne e tecnologie (SQL, XML, framework open source etc.) non si finisce mai di imparare!

Il codice di Java è inoltre Open Source, anche se oggi non è ancora chiara la strada che Oracle intende seguire in futuro. Dalla sua nascita Java è sempre stato considerato "aperto", e l'interazione con gli sviluppatori è stato sempre un punto di forza della sua evoluzione. Il sito di riferimento oggi è <http://openjdk.java.net/> ed è possibile anche attivamente contribuire allo sviluppo di Java con test, proposte e bug fixing. Le nuove caratteristiche delle ultime versioni sono tutte state proposte (utilizzando il meccanismo delle JSR) da sviluppatori che hanno dato il loro contributo di idee. Java è fatto dagli sviluppatori per gli sviluppatori.

Esistono tante altre caratteristiche che potrebbero essere menzionate, ma queste sono quelle che più dovrebbero interessare i lettori di questo libro.

1.2 Ambiente di sviluppo

In questo libro daremo per scontato che il lettore utilizzi un sistema operativo Windows 8, 7 o XP. È possibile programmare anche su altre piattaforme come Linux o Mac.

Per scrivere un programma Java, abbiamo bisogno di:

1. **Un programma che ci permetta di scrivere il codice.** Per iniziare può andar bene un semplice editor di testo, come il Blocco Note (Notepad) di Windows (sconsigliamo WordPad, Word o qualsiasi altro editor che gestisca stili, formattazioni etc.).

È anche possibile eseguire il download gratuito dell'editor Java open source EJE, agli indirizzi <http://www.claudiodesio.com/> e <http://sourceforge.net/projects/eje/>. Si tratta di un editor Java di semplice utilizzo creato da me, che offre alcune comodità rispetto ad un editor di testo generico. Tra le sue caratteristiche ricordiamo la colorazione della sintassi di Java, e il completamento di testo di alcune espressioni. Soprattutto è possibile compilare e mandare in esecuzione file, tramite la pressione di semplici pulsanti. Sono molto ben accetti feedback all'indirizzo eje@claudiodesio.com. I feedback saranno presi in considerazione per lo sviluppo delle future versioni dell'editor. Per maggiori informazioni rimandiamo le persone interessate alle pagine del sito riguardanti EJE. Tuttavia suggeriamo al lettore di utilizzare anche il Notepad "Blocco Note" (o editor equivalente) almeno per questo primo modulo. Così infatti, avrà maggiore consapevolezza degli argomenti di basso livello. È opportuno provare gli esercizi prima con il Notepad, per poi ripeterli con EJE.

2. **Il Java Development Kit versione Standard Edition** (da ora in poi **JDK**). Esso è scaricabile gratuitamente dal sito di riferimento dello sviluppatore Java: <http://www.oracle.com/technetwork/java/index.html>, con le relative note d'installazione e documentazione. Nell'appendice B è descritto il processo da seguire per procurarsi ed installare correttamente l'ambiente di sviluppo su di un sistema operativo Windows. Abbiamo

già accennato al fatto che Java è un linguaggio che si può considerare in qualche modo sia compilato che interpretato. Per iniziare abbiamo quindi bisogno di un compilatore e di una JVM. Il JDK ci offre tutto l'occorrente per lavorare in modo completo. Infatti implementa una suite di applicazioni, come un compilatore, una JVM, un formattatore di documentazione, un'altra JVM per interpretare applet, un generatore di file JAR (Java ARchive) e così via.

Si possono scaricare diverse versioni di questo software. Consigliamo comunque di scaricare la versione più recente (dalla 8 in poi).

1.2.1 Ambienti di sviluppo più complessi

Esistono diversi ambienti di sviluppo visuali più complessi che integrano editor, compilatore, ed interprete. Ne citeremo solamente due che oramai sembra si contendano la leadership assoluta: NetBeans (<http://www.netbeans.org/>) ed Eclipse (<http://www.eclipse.org/>). Notevole è il fatto che stiamo parlando di IDE open source e gratuiti!

Ognuno di questi strumenti favorisce di sicuro una velocità di sviluppo maggiore, ma per quanto riguarda il periodo d'apprendimento iniziale, è preferibile di certo scrivere tutto il codice senza gli aiuti forniti da questi strumenti, non c'è fretta. Il rischio è di non raggiungere una conoscenza "seria" di Java. Mi è capitato spesso di conoscere persone che programmavano con questi strumenti da anni, senza avere chiari alcuni concetti fondamentali. Il lettore tenga conto che se inizia a lavorare con uno dei tool di cui sopra, dovrà studiare non solo Java, ma anche il manuale del tool. Inoltre, stiamo parlando di strumenti che richiedono requisiti minimi di sistema molto alti, e per gli esercizi che verranno proposti in questo manuale, tale scelta sembra inopportuna. EJE è stato creato proprio con l'intento di evitare di utilizzare tali strumenti. Infatti esso possiede solo alcune delle comodità degli strumenti di cui sopra, ma non bisogna "studiarlo".

Intanto, iniziamo con il Blocco Note e abituiamoci da subito ad avere a che fare con più finestre aperte contemporaneamente: quelle dell'editor, e quella della prompt dei comandi (i dettagli saranno descritti nei prossimi paragrafi). Subito dopo aver provato il Blocco Note sarà possibile utilizzare EJE per valutarne l'utilità e la semplicità d'utilizzo.

1.3 Struttura del JDK

Il JDK è formato da diverse cartelle:

- ❑ **bin:** contiene tutti i file eseguibili del JDK, ovvero "javac", "java", "jar", "appletviewer", che ci permetteranno di compilare, eseguire, impacchettare etc. il nostro lavoro.
- ❑ **db:** contiene il database Java DB, una personalizzazione del database open source Apache Derby.
- ❑ **include e lib:** contengono librerie scritte in C e in Java che sono utilizzate dal JDK.
- ❑ **jre:** sta per Java Runtime Environment (JRE). Affinché un'applicazione Java risulti eseguibile su di una macchina, basta installare solo il JRE. Si tratta della JVM con il supporto per le

librerie supportate nella versione corrente di Java. Il JRE viene installato in questa cartella automaticamente, quando viene installato il JDK.

È necessario l'intero JDK però, se vogliamo sviluppare applicazioni Java, altrimenti per esempio non potremo compilare i nostri file sorgente.

- ❑ **sample:** contiene esempi (con codice sorgente) di particolari argomenti relativi a Java.
- ❑ **docs:** questa cartella deve essere scaricata ed installata a parte (cfr. appendice B), e contiene la documentazione della libreria standard di Java, più vari tutorial. Come già asserito, risulterà indispensabile.

Inoltre nella cartella principale, oltre vari file (licenza, copyright, etc.) ci sarà anche un file di nome “src.zip”. Una volta scompattato “src.zip”, sarà possibile dare uno sguardo ai file sorgenti (i “.java”) della libreria.

È bene riassumere una volta di più la differenza tra JVM, JRE e JDK.

La JVM (Java Virtual Machine, in italiano “macchina virtuale di Java”) è la macchina virtuale: il software che simula un hardware capace di interpretare ed eseguire il bytecode contenuto in un file Java compilato.

Il JRE (Java Runtime Environment, in italiano “ambiente di esecuzione di Java”) consiste in un software che mette a disposizione l’ambiente per utilizzare la JVM (e infatti la contiene). Per potere eseguire codice Java bisogna scaricare il JRE (non è possibile scaricare solo la JVM).

Il JDK (Java Development Kit, in italiano “Kit di sviluppo Java”) è il software che serve per sviluppare in Java. Contiene il JRE (e quindi la JVM) e una suite di strumenti per sviluppare. Essenzialmente, il JDK serve per sviluppare in Java, il JRE per eseguire Java.

1.3.1 Guida dello sviluppatore passo dopo passo

Dopo aver installato correttamente il JDK e impostato opportunamente le eventuali variabili di ambiente (cfr. appendice B), saremo pronti per scrivere la nostra prima applicazione (prossimo paragrafo). In generale dovremo eseguire i seguenti passi:

- 1. Scrittura del codice:** scriveremo il codice sorgente della nostra applicazione utilizzando un editor. Come già asserito precedentemente, consigliamo solo per questa prova iniziale il blocco note di Windows. Dopo aver completato la nostra prima applicazione si potrà magari riscriverla con EJE.
- 2. Salvataggio:** salveremo il nostro file con suffisso `.java`.

Se il lettore utilizza il Blocco note bisogna salvare il file chiamandolo nomeFile.java ed includendo il nome tra virgolette in questa maniera “nomeFile.java”. L'utilizzo delle virgolette non è invece necessario per salvare file con EJE. Non è necessario neanche specificare il suffisso .java di default.

- 3. Apertura prompt DOS:** una volta ottenuto il nostro file Java dobbiamo aprire una finestra Prompt di Dos (prompt dei comandi).

Se non si trova il collegamento per aprire la prompt dei comandi, per aprirla è possibile digitare il comando “cmd” nel campo esegui che si trova nel menu start di Windows. Se non siete pratici del sistema operativo DOS, nell'appendice C potete trovare un breve tutorial che vi permetterà di muovervi a vostro agio anche in questo ambiente.

- 4. Posizionamento:** Da questa prompt dobbiamo spostarci (consultare l'appendice C se non si è in grado di farlo) nella cartella in cui è stato salvato il nostro file sorgente.

- 5. Compilazione:** Eseguire il comando di compilazione:

```
javac nomeFile.java
```

Se la compilazione ha esito positivo, verrà creato un file chiamato nomeFile.class. In questo file, come abbiamo già detto, ci sarà la traduzione in bytecode del file sorgente.

- 6. Esecuzione:** a questo punto potremo mandare in esecuzione il programma invocando l'interpretazione della Java Virtual Machine. Basta scrivere dalla prompt Dos il seguente comando:

```
java nomeFile
```

(senza suffissi).

L'applicazione, a meno di errori di codice, verrà eseguita dalla JVM. Ora siamo pronti per scrivere il nostro primo programma Java.

Se si volessero ripetere questi passi utilizzando EJE, bisogna tener conto che il terzo e il quarto punto si devono sostituire rispettivamente con la pressione del tasto “compila”, e del tasto “esegui” (ci sono due passi in meno da fare).

1.4 Primo approccio al codice

Diamo subito uno sguardo alla classica applicazione “Hello World”. Si tratta del tipico primo programma che si scrive in un nuovo linguaggio di programmazione. In questo modo inizieremo a familiarizzare con la sintassi e con qualche concetto fondamentale come quello di classe e di metodo.

Avvertiamo il lettore che inevitabilmente qualche punto rimarrà oscuro, e che quindi bisognerà dare per scontate alcune parti di codice. Questo è l'approccio didattico che seguirà questo libro, invece di cercare di spiegare alcuni concetti “prematuri”, il lettore verrà a volte rimandato a moduli successivi per la loro piena comprensione. Vedremo anche come compilare e come mandare in esecuzione il nostro “mini programma”. Il fine è quello di stampare a video il messaggio “Hello World!”.

Segue il listato:

```
1 public class HelloWorld
2 {
3     public static void main(String args[])
4     {
5         System.out.println("Hello World!");
6     }
7 }
```

Questo programma deve essere salvato esattamente col nome della classe, prestando attenzione anche alle lettere maiuscole e minuscole. Questa condizione è necessaria per mandare in esecuzione l'applicazione. Il file che conterrà il listato appena presentato dovrà quindi essere chiamato “HelloWorld.java”. N.B.: I numeri non fanno parte dell'applicazione ma ci saranno utili per la sua analisi (non bisogna scriverli).

Sconsigliamo il “copia – incolla” del codice. Almeno per i primi tempi, si cerchi di scrivere tutto il codice possibile. Consigliamo al lettore di scrivere riga dopo riga dopo averne letto la seguente analisi.

1.5 Analisi del programma “HelloWorld”

Analizziamo quindi il listato precedente riga per riga:

Riga 1:

```
public class HelloWorld
```

Dichiarazione della classe `HelloWorld`. Come vedremo, ogni applicazione Java è costituita da classi. Questo concetto fondamentale sarà trattato in dettaglio nel prossimo modulo. Fare attenzione alle lettere maiuscole e minuscole. La classe è stata dichiarata accessibile in maniera pubblica mediante il modificatore `public`, ma questo concetto sarà spiegato in dettaglio nei prossimi moduli.

Riga 2:

```
{
```

Questa parentesi graffa aperta indica l'inizio della classe `HelloWorld`, che si chiuderà alla riga 7

con una parentesi graffa chiusa. Il blocco di codice compreso da queste due parentesi definisce la classe `HelloWorld`.

Sulle tastiere italiane non esiste un tasto per stampare le parentesi graffe. Possiamo però ottenerne la scrittura in diversi modi, di seguito i più utilizzati:

- ❑ tenendo premuto il tasto ALT e scrivendo 123 (per la parentesi graffa aperta) o 125 (per la parentesi graffa chiusa) con i tasti numerici che si trovano sulla destra della vostra tastiera, per poi rilasciare l'ALT;
- ❑ tenendo premuti i tasti CONTROL - SHIFT - ALT, e poi il tasto con il simbolo della parentesi quadra aperta "[" (per la parentesi graffa aperta) oppure la parentesi quadra chiusa "]" (per la parentesi graffa chiusa).

Riga 3:

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

Questa riga è bene memorizzarla da subito, poiché essa deve essere definita in ogni applicazione Java. Trattasi della dichiarazione del metodo `main()`. In Java, il termine “metodo” è sinonimo di “azione” (i metodi saranno trattati in dettaglio nel prossimo modulo). In particolare il metodo `main()` definisce il punto di partenza dell’esecuzione di ogni programma. La prima istruzione che verrà quindi eseguita in fase di esecuzione, sarà quella che la JVM troverà subito dopo l’apertura del blocco di codice che definisce questo metodo. Oltre alla parola “`main`”, la riga 3 contiene altre parole di cui studieremo in dettaglio il significato nei prossimi capitoli. Purtroppo, come già anticipato, quando si inizia a studiare un linguaggio ad oggetti come Java, è impossibile toccare un argomento senza toccarne tanti altri. Per adesso il lettore si dovrà accontentare della seguente tabella:

Termino	Spiegazione
<code>public</code>	Modificatore del metodo. I modificatori sono utilizzati in Java come nel linguaggio umano sono utilizzati gli aggettivi. Se si antepongono un modificatore alla dichiarazione di un elemento Java (un metodo, una variabile, una classe, etc.), questo cambierà in qualche modo (a seconda del significato del modificatore) le sue proprietà. In questo caso trattasi di uno specificatore d’accesso che rende di fatto il metodo accessibile anche al di fuori della classe in cui è stato definito.
<code>static</code>	Altro modificatore del metodo. La definizione di <code>static</code> è abbastanza complessa. Per ora il lettore si accontenti di sapere che è essenziale per la definizione del metodo <code>main()</code> .
<code>void</code>	È il tipo di ritorno del metodo. Significa “vuoto” e quindi questo metodo non restituisce nessun tipo di valore. Il metodo <code>main()</code> non deve mai avere un tipo di ritorno diverso da <code>void</code> .
<code>main</code>	Trattasi del nome del metodo (detto anche “identificatore” del metodo).
<code>(String args[])</code>	Alla destra dell’identificatore di un metodo, si definisce sempre una coppia di parentesi tonde che racchiude opzionalmente una lista di parametri (detti anche argomenti del metodo). Il metodo <code>main()</code> , in ogni caso, vuole sempre come parametro un array di stringhe (agli array troveremo dedicata nel terzo modulo un’intera unità didattica). Si noti che <code>args</code> è l’identificatore (nome) dell’array, ed è l’unica parola che può variare nella definizione del metodo <code>main()</code> , anche se per convenzione si utilizza sempre <code>args</code> .

Riga 4:

{

Questa parentesi graffa indica l'inizio del metodo `main()` che si chiuderà alla riga 6 con una parentesi graffa chiusa. Il blocco di codice compreso tra queste due parentesi definisce il metodo.

Riga 5:

```
System.out.println("HelloWorld!");
```

Questo comando stamperà a video la stringa "Hello World!". Anche in questo caso, giacché dovremmo introdurre argomenti per i quali il lettore non è ancora maturo, preferiamo rimandare la spiegazione dettagliata di questo comando ai prossimi capitoli. Per ora ci basterà sapere che stiamo invocando un metodo appartenente alla libreria standard di Java che si chiama `println()`, passandogli come parametro la stringa che dovrà essere stampata.

Riga 6:

}

Questa parentesi graffa chiusa chiude l'ultima che è stata aperta, ovvero chiude il blocco di codice che definisce il metodo `main()`.

Riga 7:

}

Questa parentesi graffa invece chiude il blocco di codice che definisce la classe `HelloWorld`. Si noti che le parentesi erano state indentate in maniera non casuale.

1.6 Compilazione ed esecuzione del programma `HelloWorld`

Una volta riscritto il listato utilizzando il blocco note dobbiamo salvare il nostro file in una cartella di lavoro, chiamata ad esempio "CorsoJava".

Il lettore presti attenzione al momento del salvataggio. Essendo il blocco note un editor di testo generico, e non un editor per Java, questo tende a salvare i file con il suffisso ".txt". Bisogna quindi includere il nome del file (che ricordiamo deve essere obbligatoriamente `HelloWorld.java`) tra virgolette in questo modo "`HelloWorld.java`".

L'utilizzo delle virgolette non è necessario per salvare file con EJE che è invece un editor per Java. Non è necessario neanche specificare il suffisso .java di default.

A questo punto possiamo iniziare ad aprire una prompt di Dos e a spostarci all'interno della nostra cartella di lavoro. Dopo esserci accertati che il file `HelloWorld.java` esista, possiamo passare alla

fase di compilazione. Se eseguiamo il comando:

```
javac HelloWorld.java
```

diamo il nostro file sorgente in input al compilatore che mette a disposizione il JDK. Se al termine della compilazione non viene fornito nessun messaggio d'errore, significa che la compilazione ha avuto successo. A questo punto possiamo notare che nella nostra cartella di lavoro è stato creato un file di nome `HelloWorld.class`. Questo è appunto il file sorgente tradotto in bytecode, pronto per essere interpretato dalla JVM. Se quindi eseguiamo il comando:

```
java HelloWorld
```

il nostro programma, se non sono sollevate eccezioni dalla JVM, sarà eseguito stampando il tanto sospirato messaggio.

Si consiglia di ripetere questi passi con le semplificazioni che ci offre EJE.

1.7 Possibili problemi in fase di compilazione ed esecuzione

Di solito, i primi tempi, gli aspiranti programmati Java ricevono spesso dei messaggi apparentemente misteriosi da parte del compilatore e dell'interprete Java. Non bisogna scoraggiarsi! Bisogna avere pazienza e imparare a leggere i messaggi che vengono restituiti. Inizialmente può sembrare difficile, ma in breve tempo ci si accorge che gli errori che si commettono sono spesso gli stessi.

1.7.1 Possibili messaggi di errore in fase di compilazione

```
javac: Command not found
```

In questo caso non è il compilatore che ci sta segnalando un problema, bensì, è lo stesso sistema operativo che non riconosce il comando `javac` che dovrebbe chiamare il compilatore del JDK. Probabilmente quest'ultimo non è stato installato correttamente. Un tipico problema è di non aver impostato la variabile d'ambiente PATH (cfr. appendice B).

```
HelloWorld.java:3: error: cannot find symbol HelloWorld.java:1: error:  
class Helloworld is public, should be declared in a file named  
Helloworld.java  
public class Helloworld{  
^  
1 error
```

In questo caso abbiamo chiamato il file “HelloWorld” e la classe “Helloworld” (notare la “w”)

minuscola). Il compilatore non è così intelligente da capire che non l'abbiamo fatto volontariamente.

```
HelloWorld.java:3: error: cannot find symbol
    System.out.println("Hello World!");
          ^
symbol:  method println(String)
location: variable out of type PrintStream
1 error
```

Se riceviamo questo messaggio, abbiamo semplicemente scritto “`println`” in luogo di “`println`”. Il compilatore non può da solo accorgersi che è stato semplicemente un errore di battitura, quindi ha segnalato che il metodo `println()` non è stato trovato. In fase di debug è sempre bene prendere coscienza dei messaggi di errore che fornisce il compilatore, tenendo ben presente però, che ci sono dei limiti a questi messaggi ed alla comprensione degli errori del compilatore stesso.

```
HelloWorld.java:1: error: class, interface, or enum expected
public Class HelloWorld{
          ^
HelloWorld.java:2: error: class, interface, or enum expected
    public static void main (String args[]){
          ^
HelloWorld.java:4: error: class, interface, or enum expected
    }
    ^
3 errors
```

In questo caso invece avremo scritto `class` con lettera maiuscola e quindi la JVM ha richiesto esplicitamente una dichiarazione di classe (o di interfaccia o enum, concetti che chiariremo più avanti). “`Class`” lo ripetiamo, non è la stessa cosa di “`class`” in Java.

```
HelloWorld.java:3: error: ';' expected
System.out.println("Hello World!")
          ^
1 error
```

In questo caso il compilatore ha capito subito che abbiamo dimenticato un punto e virgola, che serve a concludere ogni statement (istruzione). Purtroppo il nostro compilatore non sarà sempre così preciso. In alcuni casi, se dimentichiamo un punto e virgola, o peggio dimentichiamo di chiudere un blocco di codice con una parentesi graffa, il compilatore potrebbe segnalare l'esistenza di errori inesistenti in righe di codice successive.

1.7.2 Possibili messaggi relativi alla fase di interpretazione

Solitamente in questa fase vengono sollevate dalla JVM quelle che vengono definite

come “eccezioni”. Alle eccezioni sarà dedicato un intero modulo più avanti.

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Se questo è il messaggio di risposta ad un tentativo di mandare in esecuzione un programma, abbiamo probabilmente definito male il metodo `main()`. Probabilmente abbiamo dimenticato di dichiararlo `static` o `public`, oppure abbiamo scritto male la lista degli argomenti (che deve essere `String args[]`), o magari non abbiamo chiamato “`main`” il metodo.

```
Errore: impossibile trovare o caricare la classe principale HelloWorld.class
```

Anche in questo caso è il sistema operativo che sta segnalando l’errore. In questo caso abbiamo provato a eseguire il comando:

“`java HelloWorld.class`” invece di “`java HelloWorld`”.

È possibile che abbiate installato sulla vostra macchina programmi che impostano la variabile del sistema operativo CLASSPATH (per esempio Oracle). In tal caso non è più possibile eseguire i nostri file con il comando:

```
java HelloWorld
```

ma bisogna aggiungere la seguente opzione:

```
java -classpath . HelloWorld
```

Con EJE invece è possibile compilare ed eseguire l’applicativo senza nessun lavoro extra. Alla gestione della variabile CLASSPATH è dedicata l’appendice E (e per ora non è una priorità).

Riepilogo

In questo modulo abbiamo definito **Java** come linguaggio e come tecnologia e abbiamo descritto il linguaggio mediante le sue caratteristiche. Abbiamo inoltre descritto **l’ambiente di sviluppo** ed abbiamo imparato a **compilare e mandare in esecuzione** una semplice applicazione Java. Abbiamo approcciato ad un piccolo programma per avere un’idea di alcuni concetti fondamentali e prendere confidenza con il codice e l’ambiente di sviluppo di base. Infine abbiamo anche posto l’accento sull’importanza di leggere **i messaggi di errore** che il compilatore o la JVM mostrano allo sviluppatore.

Esercizi modulo 1

Esercizio 1.a)

Digitare, salvare, compilare ed eseguire il programma `HelloWorld`. Consigliamo al lettore di eseguire questo esercizio due volte: la prima volta utilizzando il Notepad e la prompt Dos, e la seconda utilizzando EJE.

EJE permette di inserire parti di codice pre-formattate tramite il menu “Inserisci” (o tramite short-cut).

Esercizio 1.b) Caratteristiche di Java, Vero o Falso:

1. Java è il nome di una tecnologia e contemporaneamente il nome di un linguaggio di programmazione.
2. Java è un linguaggio interpretato ma non compilato.
3. Java è un linguaggio veloce ma non robusto.
4. Java è un linguaggio difficile da imparare perché in ogni caso obbliga ad imparare l'object orientation.
5. La Java Virtual Machine è un software che supervisiona il software scritto in Java.
6. La JVM gestisce la memoria automaticamente mediante la Garbage Collection.
7. L'indipendenza dalla piattaforma è una caratteristica poco importante.
8. Java è un sistema chiuso.
9. La Garbage Collection garantisce l'indipendenza dalla piattaforma.
10. Java è un linguaggio gratuito che raccoglie le caratteristiche migliori di altri linguaggi, ed esclude quelle ritenute peggiori e più pericolose.

Esercizio 1.c) Codice Java, Vero o Falso:

1. La seguente dichiarazione del metodo `main()` è corretta:

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

2. La seguente dichiarazione del metodo `main()` è corretta:

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

3. La seguente dichiarazione del metodo `main()` è corretta:

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

4. La seguente dichiarazione del metodo `main()` è corretta:

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

5. La seguente dichiarazione di classe è corretta:

```
public class {...}
```

6. La seguente dichiarazione di classe è corretta:

```
public Class Auto {...}
```

7. La seguente dichiarazione di classe è corretta:

```
public class Auto {...}
```

8. È possibile dichiarare un metodo al di fuori del blocco di codice che definisce una classe.

9. Il blocco di codice che definisce un metodo è delimitato da due parentesi tonde.

10. Il blocco di codice che definisce un metodo è delimitato da due parentesi quadre.

Esercizio 1.d) Ambiente e processo di sviluppo, Vero o Falso:

1. La JVM è un software che simula un hardware.
2. Il bytecode è contenuto in un file con suffisso “.class”.
3. Lo sviluppo Java consiste nello scrivere il programma, salvarlo, mandarlo in esecuzione ed infine compilarlo.
4. Lo sviluppo Java consiste nello scrivere il programma, salvarlo, compilarlo ed infine mandarlo in esecuzione.
5. Il nome del file che contiene una classe Java deve coincidere con il nome della classe, anche se non si tiene conto delle lettere maiuscole e minuscole.
6. Una volta compilato un programma scritto in Java è possibile eseguirlo su di un qualsiasi sistema operativo che abbia una JVM.
7. Per eseguire una qualsiasi applicazione Java basta avere un browser.
8. Il compilatore del JDK viene invocato tramite il comando `javac` e la JVM viene invocata tramite il comando `java`.
9. Per mandare in esecuzione un file che si chiama `pippo.class` dobbiamo eseguire il seguente comando dalla prompt: `java pippo.java`.
10. Per mandare in esecuzione un file che si chiama `pippo.class` dobbiamo eseguire il seguente comando dalla prompt: `java pippo.class`.

Soluzioni esercizi modulo 1

Esercizio 1.a)

Non è prevista una soluzione per questo esercizio.

Esercizio 1.b) Caratteristiche di Java, Vero o Falso:

- 1. Vero.**
- 2. Falso.**
- 3. Falso.**
- 4. Vero.**
- 5. Vero.**
- 6. Vero.**
- 7. Falso.**
- 8. Falso.**
- 9. Falso.**
- 10. Vero.**

Esercizio 1.c) Codice Java, Vero o Falso:

- 1. Falso**, manca il tipo di ritorno (`void`).
- 2. Falso**, l'identificatore dovrebbe iniziare con lettera minuscola (`main`).
- 3. Vero.**
- 4. Vero.**
- 5. Falso**, manca l'identificatore.
- 6. Falso**, la parola chiave si scrive con lettera iniziale minuscola (`class`).
- 7. Vero.**
- 8. Falso.**
- 9. Falso**, le parentesi sono graffe.
- 10. Falso**, le parentesi sono graffe.

Esercizio 1.d) Ambiente e processo di sviluppo, Vero o Falso:

- 1. Vero.**
- 2. Vero.**
- 3. Falso**, bisogna prima compilarlo per poi mandarlo in esecuzione.
- 4. Vero.**
- 5. Falso**, bisogna anche tenere conto delle lettere maiuscole e minuscole.
- 6. Vero.**
- 7. Falso**, un browser è sufficiente solo per eseguire applet.
- 8. Vero.**
- 9. Falso**, il comando corretto è “`java pippo`”.
- 10. Falso**, il comando corretto è “`java pippo`”.

**Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper definire il linguaggio di programmazione Java e le sue caratteristiche. (unità 1.1)	<input type="checkbox"/>	
Interagire con l'ambiente di sviluppo: il Java Development Kit. (unità 1.2, 1.3, 1.6, 1.7)	<input type="checkbox"/>	
Saper digitare, compilare e mandare in esecuzione una semplice applicazione. (unità 1.4, 1.5)	<input type="checkbox"/>	

Note:

Componenti fondamentali di un programma Java

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Saper definire i concetti di classe, oggetto, variabile, metodo e costruttore (unità 2.1, 2.2, 2.3, 2.4, 2.5).
- ✓ Saper dichiarare una classe (unità 2.1).
- ✓ Istanziare oggetti da una classe (unità 2.2).
- ✓ Utilizzare i membri pubblici di un oggetto sfruttando l'operatore dot (unità 2.2, 2.3, 2.4).
- ✓ Dichiarare ed invocare un metodo (unità 2.3).
- ✓ Saper dichiarare e inizializzare una variabile (unità 2.4).
- ✓ Saper definire ed utilizzare i diversi tipi di variabili (d'istanza, locali e parametri formali) (unità 2.4).
- ✓ Dichiarare ed invocare un metodo costruttore (unità 2.5).
- ✓ Saper definire un costruttore di default (unità 2.5).
- ✓ Saper accennare alla definizione di package (unità 2.6).

Da questo modulo parte lo studio del linguaggio vero e proprio. Dapprima familiarizzeremo con i concetti fondamentali che sono alla base di un programma Java, sia in maniera teorica che pratica.

Anche in questo caso, non daremo nulla per scontato e quindi anche il lettore che non ha mai programmato dovrebbe poter iniziare.

I lettori che invece hanno esperienza con la programmazione funzionale, troveranno profonde differenze con i concetti dell'object orientation. Il consiglio per questi ultimi è di non ancorarsi troppo a quello che già si conosce. È inutile per esempio forzare una classe ad avere il ruolo che poteva avere una funzione nella programmazione funzionale. Meglio far finta di partire da zero! Per chi invece già conosce altri linguaggi orientati agli oggetti, questo modulo dovrebbe risultare chiaro e semplice.

2.1 Componenti fondamentali di un programma Java

Ecco una lista di concetti che sono alla base della conoscenza di Java:

- ❑ Classe
- ❑ Oggetto
- ❑ Membro:
 - ❑ Attributo (variabile membro)

- Metodo (metodo membro)
- Costruttore
- Package

Esistono altri componenti importanti per programmare in Java, come per esempio le interfacce, le annotazioni, le classi innestate e le enumerazioni. Per metodo didattico si è scelto di evitare di introdurre nozioni che attualmente non sono necessarie per iniziare a programmare. Questi altri concetti verranno presentati durante la lettura di questo libro nel momento in cui il lettore dovrebbe essere pronto per apprezzarne l'utilizzo. Di seguito vengono quindi fornite al lettore solo le nozioni fondamentali per approcciare nel modo migliore alle caratteristiche del linguaggio. Prima però, introduciamo una convenzione ausiliaria per chi non ha confidenza con la programmazione ad oggetti.

2.1.1 Convenzione per la programmazione Java

L'apprendimento di un linguaggio orientato agli oggetti può essere spesso molto travagliato, specialmente se si possiedono solide "radici funzionali". Abbiamo già accennato al fatto che Java, a differenza del C++, in pratica "ci obbliga" a programmare ad oggetti. Non ha senso imparare il linguaggio senza sfruttare il supporto che esso offre alla programmazione ad oggetti. Chi impara il C++ ha un vantaggio illusorio. Può infatti permettersi di continuare ad utilizzare il paradigma funzionale, e quindi imparare il linguaggio insieme ai concetti object oriented, ma mettendo inizialmente questi ultimi da parte. Infatti, in C++, è comunque possibile creare funzioni e programmi chiamanti. Essendo C++ un'estensione di C, lo si può utilizzare allo stesso modo in cui si utilizzava il C, in maniera funzionale. In realtà si tratta di un vantaggio a breve termine, che si traduce inevitabilmente in uno svantaggio a lungo termine. Con Java lo scenario si presenta più complesso. Inizieremo direttamente a creare applicazioni che saranno costituite da un certo numero di classi. Non basterà dare per scontato che "Java funziona così". Bisognerà invece cercare di chiarire ogni singolo punto poco chiaro, in modo tale da non creare troppa confusione, che, a lungo andare, potrebbe scoraggiare. Per fortuna Java è un linguaggio con caratteristiche molto chiare e coerenti, e questo ci sarà di grande aiuto. Alla fine l'aver appreso in modo profondo determinati concetti dovrebbe regalare al lettore maggiore soddisfazione.

Proviamo da subito ad approcciare il codice in maniera schematica, introducendo una convenzione.

In questo testo distingueremo due ambiti quando sviluppiamo un'applicazione Java:

1. Ambito del compilatore (o compilativo, o delle classi)
2. Ambito della Java Virtual Machine (o esecutivo, o degli oggetti)

All'interno dell'ambito del compilatore codificheremo la parte strutturale della nostra applicazione. L'ambito esecutivo invece ci servirà per definire il flusso di lavoro che dovrà essere eseguito.

Distinguere questi due ambiti sarà utile per comprendere i ruoli che dovranno avere all'interno delle nostre applicazioni i concetti introdotti in questo modulo.

2.2 Le basi della programmazione object oriented: classi ed oggetti

I concetti di classe ed oggetto sono strettamente legati. Esistono infatti definizioni ufficiali derivanti dalla teoria della programmazione orientata agli oggetti, che di seguito presentiamo:

- Definizione 1: una classe è un'astrazione indicante un insieme di oggetti che condividono le stesse caratteristiche e le stesse funzionalità.
- Definizione 2: un oggetto è un'istanza (ovvero una creazione fisica) di una classe.

A questo punto il lettore più “matematico” sicuramente avrà le idee un po’ confuse. Effettivamente definire il concetto di classe tramite la definizione di oggetto, e il concetto di oggetto tramite la definizione di classe, non è certo il massimo della chiarezza! La situazione però è molto meno complicata di quella che può sembrare. Ogni concetto che fa parte della teoria dell’Object Orientation infatti, esiste anche nel mondo reale. Questa teoria è nata proprio per soddisfare l’esigenza umana di interagire con il software in maniera più naturale (cfr. paragrafo 5.1). Si tratterà quindi solo di associare la giusta idea al nuovo termine. Passiamo subito ad un esempio:

```
public class Punto
{
    public int x;
    public int y;
}
```

Con Java è possibile creare codice per astrarre un qualsiasi concetto del mondo reale. Abbiamo appena definito una classe `Punto`. Evidentemente lo scopo di questa classe è definire il concetto di punto (a due dimensioni) tramite la definizione delle sue coordinate su di un piano cartesiano. Le coordinate sono state astratte mediante due attributi (variabili) dichiarati di tipo intero (`int`) e pubblici (`public`), chiamati `x` ed `y`. Questa classe potrebbe costituire una parte di un’applicazione Java, per esempio un’applicazione di disegno.

Possiamo salvare questo listato in un file chiamato `Punto.java`, e compilarlo tramite il comando

```
javac Punto.java
```

ottenendo così il file `Punto.class`.

Non possiamo però mandarlo in esecuzione tramite il comando

```
java Punto
```

Ottureremo infatti un messaggio di errore. Questo perché in questa classe è assente il metodo `main()` (cfr. modulo precedente), che abbiamo definito come punto di partenza dell’esecuzione di ogni applicazione Java. Se eseguissimo il comando `java Punto`, la JVM cercherebbe questo metodo per eseguirlo, e non trovandolo terminerebbe istantaneamente l’esecuzione. Ciò è assolutamente in linea con la definizione di classe (def. 1). Infatti, se volessimo trovare nel mondo reale un sinonimo di classe, dovremmo pensare a termini come “idea”, “astrazione”, “concetto”, “modello” o “definizione”. Quindi, definendo una classe `Punto` con codice Java, abbiamo solo definito all’interno

del nostro futuro programma il concetto di punto, secondo la nostra interpretazione e nel contesto in cui ci vogliamo calare (per esempio un programma di disegno). Con il codice scritto finora in realtà, non abbiamo ancora definito nessun punto vero e proprio (per esempio il punto di coordinate x=5 e y=6).

Quindi, come nel mondo reale, se non esistono punti concreti (ma solo la definizione di punto) non può accadere nulla che abbia a che fare con un punto e così, nel codice Java, la classe Punto non è eseguibile da sola!

Occorre quindi definire gli oggetti punto, ovvero le creazioni fisiche realizzate a partire dalla definizione data dalla classe.

Nel contesto della programmazione ad oggetti, una classe dovrebbe quindi limitarsi a definire che struttura avranno gli oggetti che da essa saranno istanziati. **Istanziare**, come abbiamo già affermato, è il termine object oriented che si usa in luogo di “creare fisicamente”, ed il risultato di un’istanziazione viene chiamato **oggetto**.

Per esempio, istanzieremo oggetti dalla classe Punto creando un’altra classe che contiene un metodo main():

```
1 public class Principale
2 {
3     public static void main(String args[])
4     {
5         Punto punto1;
6         punto1 = new Punto();
7         punto1.x = 2;
8         punto1.y = 6;
9         Punto punto2 = new Punto();
10        punto2.x = 0;
11        punto2.y = 1;
12        System.out.println(punto1.x);
13        System.out.println(punto1.y);
14        System.out.println(punto2.x);
15        System.out.println(punto2.y);
16    }
17 }
```

Analizziamo la classe Principale.

Alla riga 5 è dichiarato un oggetto di tipo Punto che viene chiamato punto1:

```
5     Punto punto1;
```

Ma è solamente alla riga 6 che avviene l’istanziazione (creazione) della classe Punto:

```
6     punto1 = new Punto();
```

Di fatto la parola chiave `new` istanzia la classe `Punto` a cui viene assegnato il nome `punto1`. Dalla riga 6 in poi è possibile utilizzare l'oggetto `punto1`. Precisamente, alle righe 7 e 8, assegniamo alle coordinate `x` e `y` del `punto1` rispettivamente i valori interi 2 e 6:

```
7     punto1.x = 2;  
8     punto1.y = 6;
```

In pratica, sfruttando la definizione che ci ha fornito la classe `Punto`, abbiamo creato un oggetto di tipo `Punto` che è individuato dal nome `punto1`.

Notiamo l'utilizzo dell'operatore “dot” (che in inglese significa “punto”, ma nel senso del simbolo di punteggiatura “.”) per accedere alle variabili `x` e `y`.

Ricapitolando, abbiamo prima dichiarato l'oggetto alla riga 5, l'abbiamo istanziato alla riga 6, e l'abbiamo utilizzato (impostando le sue variabili) alle righe 7 e 8.

Alla riga 9 poi, abbiamo dichiarato ed istanziato con un'unica riga di codice un altro oggetto dalla classe `Punto`, chiamandolo `punto2`:

```
9     Punto punto2 = new Punto();
```

Abbiamo poi impostato le coordinate di quest'ultimo a 0 e 1:

```
10    punto2.x = 0;  
11    punto2.y = 1;
```

Abbiamo infine stampato le coordinate di entrambi i punti:

```
12    System.out.println(punto1.x);  
13    System.out.println(punto1.y);  
14    System.out.println(punto2.x);  
15    System.out.println(punto2.y);
```

Le definizioni di classi ed oggetto dovrebbero essere un po' più chiare: la classe è servita per definire come saranno fatti gli oggetti. L'oggetto rappresenta una realizzazione fisica della classe.

In questo esempio abbiamo istanziato due oggetti diversi da una stessa classe. Entrambi questi oggetti sono punti, ma evidentemente sono punti diversi (si trovano in diverse posizioni sul riferimento cartesiano).

Questo ragionamento, in fondo, è già familiare a chiunque, giacché derivante dal mondo reale che ci circonda. L'essere umano, per superare la complessità della realtà, raggruppa gli oggetti in classi. Per esempio, nella nostra mente esiste il modello definito dalla classe `Persona`. Ma nella realtà esistono miliardi di oggetti di tipo `Persona`, ognuno dei quali ha caratteristiche uniche.

Comprendere le definizioni 1 e 2, ora non dovrebbe rappresentare più un problema.

2.2.1 Osservazione importante sulla classe Punto

Nell'esempio precedente abbiamo potuto commentare la definizione di due classi. Per la prima (la classe `Punto`) abbiamo sottolineato la caratteristica di rappresentare *un dato*. È da considerarsi una parte strutturale dell'applicazione e quindi svolge un ruolo essenziale nell'ambito compilativo. Nell'ambito esecutivo, la classe `Punto` non ha un ruolo attivo. Infatti sono gli oggetti istanziati da essa che influenzano il flusso di lavoro del programma. Questo può definirsi come il caso standard. In un'applicazione object oriented, una classe dovrebbe limitarsi a definire la struttura comune di un insieme di oggetti, e non dovrebbe mai possedere né variabili né metodi. Infatti, la classe `Punto` non possiede le variabili `x` e `y`, bensì, dichiarando le due variabili, definisce gli oggetti che da essa saranno istanziati come possessori di quelle variabili. Si noti che mai all'interno del codice dell'ambito esecutivo è presente un'istruzione come:

```
Punto.x
```

ovvero

```
NomeClasse.nomeVariabile
```

(che tra l'altro produrrebbe un errore in compilazione) bensì:

```
punto1.x
```

ovvero

```
nomeOggetto.nomeVariabile
```

L'operatore “.” è sinonimo di “appartenenza”. Sono quindi gli oggetti a possedere le variabili dichiarate nella classe (che tra l'altro verranno dette “variabili d'istanza”, ovvero dell'oggetto). Infatti, i due oggetti istanziati avevano valori diversi per `x` e `y`, il che significa che l'oggetto `punto1` e l'oggetto `punto2` hanno le proprie variabili `x` e `y`. In altre parole le variabili di `punto1` sono assolutamente indipendenti dalle variabili di `punto2`. Giacché le classi non hanno membri (variabili e metodi), non eseguono codice e non hanno un ruolo nell'ambito esecutivo, e per quanto visto sono gli oggetti protagonisti assoluti di quest'ambito.

2.2.2 Osservazione importante sulla classe Principale

Come spesso accade quando si approccia Java, appena definita una nuova regola, subito si presenta l'eccezione (che se vogliamo... dovrebbe confermarla!). Puntiamo infatti la nostra attenzione sulla classe `Principale`. È una classe che esegue codice contenuto all'interno dell'unico metodo `main()`,

che per quanto detto è assunto per default quale punto di partenza di un'applicazione Java. In qualche modo infatti, i creatori di Java dovevano stabilire un modo per far partire l'esecuzione di un programma. La scelta è stata compiuta in base ad una questione pratica e storica: un'applicazione scritta in C o C++ ha come punto di partenza per default un programma chiamante che si deve nominare proprio `main()`. In Java i programmi chiamanti non esistono, ma esistono i metodi, ed in particolare i metodi `statici`, ovvero dichiarati con un modificatore static. Questo modificatore sarà trattato in dettaglio nel modulo 6. Per ora limitiamoci a sapere che un membro dichiarato statico appartiene alla classe, e che tutti gli oggetti istanziati da essa condivideranno i membri statici. Concludendo, giacché la classe Principale contiene il metodo `main()`, può eseguire codice.

In ogni applicazione Java deve esserci una classe che contiene il metodo `main()`. Questa classe dovrebbe avere il nome dell'applicazione stessa, astraendo il flusso di lavoro che deve essere eseguito. In linea teorica, quindi, la classe contenente il metodo `main()` non dovrebbe contenere altri membri.

2.2.3 Un'altra osservazione importante

Abbiamo presentato l'esempio precedente per la sua semplicità. È infatti matematico pensare ad un punto sul piano cartesiano come formato da due coordinate chiamate `x` e `y`. Ci sono concetti del mondo reale la cui astrazione non è così standard. Per esempio, volendo definire l'idea (astrazione) di un'auto, dovremmo parlare delle caratteristiche e delle funzionalità comuni ad ogni auto. Ma, se confrontiamo l'idea (ovvero la classe) di auto che definirebbe un intenditore di automobili con quella di una persona che non ha neanche la patente, ci sarebbero significative differenze. La persona non patentata potrebbe definire un'auto come "un mezzo di trasporto (quindi che si può muovere) con quattro ruote ed un motore", ma l'intenditore di automobili potrebbe dare una definizione alternativa molto più dettagliata definendo caratteristiche come assicurazione, telaio, modello, pneumatici etc. Entrambe queste definizioni potrebbero essere portate nella programmazione sotto forma di classi. Per convenienza ragioneremo "come se non avessimo la patente", creando una classe dichiarante come attributo un intero chiamato `numeroRuote` inizializzato a 4. Inoltre per semplicità definiamo un'altra variabile intera `cilindrata` (in luogo del motore) ed un metodo che potremmo chiamare `muoviti()`.

Nei prossimi moduli i metodi saranno definiti in dettagli. Per adesso ci basterà sapere che il metodo serve a definire una funzionalità che deve avere il concetto che si sta astraendo con la classe.

```
public class Auto
{
    public int numeroRuote = 4;
    public int cilindrata; // quanto vale?
    public void muoviti()
```

```
{  
    // implementazione del metodo...  
}  
}
```

Il codice che scriviamo dopo due slash (come nel metodo `muoviti()`), è un commento, e non verrà preso in considerazione dal compilatore. I commenti di Java verranno comunque spiegati in dettaglio nel prossimo modulo.

Ogni auto ha 4 ruote (di solito), si muove ed ha una cilindrata (il cui valore non è definibile a priori). Una Ferrari California e una Fiat 600 hanno entrambe 4 ruote, una cilindrata e si muovono, anche se in modo diverso. La Ferrari e la Fiat sono da considerarsi oggetti della classe `Auto`, e nella realtà esisterebbero come oggetti concreti. Nella seguente classe vengono creati due oggetti della classe `Auto`, viene impostata la loro cilindrata opportunamente, e per entrambi gli oggetti verrà invocato il metodo `muoviti()`.

```
1 public class Principale2  
2 {  
3     public static void main(String args[])  
4     {  
5         Auto fiat600;  
6         fiat600 = new Auto();  
7         fiat600.cilindrata = 1100;  
8         fiat600.muoviti();  
9         Auto california = new Auto();  
10        california.cilindrata = 4300;  
11        california.muoviti();  
12    }  
13 }
```

Dovrebbe essere abbastanza chiaro che i due oggetti si muoveranno entrambi, ma a velocità diverse!

2.3 I metodi in Java

Nella definizione di classe, quando si parla di *caratteristiche* ci si riferisce ai **dati** (variabili e costanti), mentre col termine *funzionalità* ci si riferisce ai **metodi**. Abbiamo già accennato al fatto che il termine **metodo** è sinonimo di *azione*. Quindi, affinché un programma esegua qualche istruzione deve contenere metodi. Per esempio è il metodo `main()` che, per default, è il punto di partenza di ogni applicazione Java. Una classe senza metodo `main()`, come la classe `Punto`, non può essere eseguita ma solo istanziata all'interno di un metodo di un'altra classe (nell'esempio precedente, nel metodo `main()` della classe `Principale`). Il concetto di metodo è quindi anch'esso alla base della programmazione ad oggetti. Senza metodi, gli oggetti non potrebbero comunicare tra loro. Inoltre, i metodi rendono i programmi più leggibili e di più facile manutenzione, lo sviluppo più veloce e stabile, evitano le duplicazioni e favoriscono il riuso del software.

Il programmatore che si avvicina a Java dopo esperienze nel campo della programmazione funzionale, spesso tende a confrontare il concetto di funzione con il concetto di metodo. Sebbene simili nella forma e nella sostanza, bisognerà tener presente, che un metodo ha un ruolo differente rispetto ad una funzione. Nella programmazione strutturata, infatti, il concetto di funzione era alla base. Tutti i programmi erano formati da un programma chiamante e da un certo numero di funzioni. Queste avevano di fatto il compito di risolvere determinati "sotto-problemi" generati da un'analisi di tipo top-down, allo scopo di risolvere il problema generale. Come vedremo più avanti, nella programmazione orientata agli oggetti, i "sotto-problemi" saranno invece risolti tramite l'astrazione di classi ed oggetti, che a loro volta definiranno metodi.

È bene che il lettore cominci a distinguere nettamente due fasi per quanto riguarda i metodi: dichiarazione e chiamata (ovvero la definizione e l'utilizzo).

2.3.1 Dichiarazione di un metodo

La dichiarazione definisce un metodo. Ecco la sintassi:

```
[modificatori] tipo_di_ritorno nome_del_metodo ([parametri])  
{corpo_del_metodo}
```

dove:

- **modificatori:** parole chiave di Java che possono essere usate per modificare in qualche modo le funzionalità e le caratteristiche di un metodo. Tutti i modificatori sono trattati in maniera approfondita nei prossimi moduli. Per ora ci accontenteremo di conoscerne superficialmente solo alcuni. Esempi di modificatori sono `public` e `static`;
- **tipo di ritorno:** il tipo di dato che un metodo potrà restituire dopo essere stato chiamato. Questo potrebbe coincidere sia con un tipo di dato primitivo come un `int`, sia con un tipo complesso (un oggetto) come una stringa (definita dalla classe `String`). È anche possibile specificare che un metodo non restituisca nulla (`void`), come nel caso del `main()`;
- **nome del metodo:** identificatore del metodo;
- **parametri:** dichiarazioni di variabili che potranno essere passate al metodo, e di conseguenza essere sfruttate nel corpo del metodo al momento della chiamata. Il numero di parametri può essere zero, ma anche maggiore di uno. Se si dichiarano più parametri le loro dichiarazioni saranno separate da virgole;
- **corpo del metodo:** insieme di istruzioni (statement) che verranno eseguite quando il metodo sarà invocato.

La coppia costituita dal nome del metodo e l'eventuale lista dei parametri viene detta

“firma” (in inglese “signature”) del metodo.

Per esempio, viene di seguito presentata una classe chiamata `Aritmetica` che dichiara un banale metodo di somma di due numeri:

```
public class Aritmetica
{
    public int somma(int a, int b)
    {
        return (a + b);
    }
}
```

Si noti che il metodo presenta come modificatore la parola chiave `public`. Si tratta di uno specificatore (o modificatore) d’accesso che rende il metodo `somma()` accessibile da altre classi. Precisamente, i metodi di altre classi potranno, dopo aver istanziato un oggetto della classe `Aritmetica`, invocare il metodo `somma()`. Esistono anche altri specificatori d’accesso come `private` ed approfondiremo il discorso nel modulo 5.

Il tipo di ritorno è un `int`, ovvero un intero. Ciò significa che questo metodo avrà come ultima istruzione un comando (`return`) che restituirà un numero intero. Inoltre la dichiarazione di questo metodo evidenzia come lista di parametri due interi (chiamati `a` e `b`). Questi saranno sommati all’interno del blocco di codice e la loro somma verrà restituita, tramite il comando `return`, come risultato finale. La somma tra due numeri interi non può essere che un numero intero. Concludendo, la dichiarazione di un metodo definisce quali azioni deve compiere il metodo stesso, quando sarà chiamato.

Si noti che nell’esempio sono state utilizzate le parentesi tonde per circondare l’operazione di somma. Anche se in questo caso non erano necessarie per la corretta esecuzione del metodo, abbiamo preferito utilizzarle per rendere più chiara l’istruzione.

2.3.2 Chiamata (o invocazione) di un metodo

Di seguito presentiamo un’altra classe *eseguibile* (ovvero contenente il metodo `main()`), che istanzia un oggetto dalla classe `Aritmetica` e chiama il metodo `somma()`:

```
1 public class Uno
2 {
3     public static void main(String args[])
4     {
5         Aritmetica oggetto1 = new Aritmetica();
6         int risultato = oggetto1.somma(5, 6);
7     }
8 }
```

In questo caso notiamo subito che l'accesso al metodo `somma()` è avvenuto sempre tramite l'operatore `dot` come nel caso dell'accesso alle variabili. Quindi, tutti i membri (attributi e metodi) pubblici definiti all'interno di una classe saranno accessibili tramite un'istanza della classe stessa che sfrutta l'operatore `dot`. La sintassi da utilizzare è la seguente:

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

e

```
nomeOggetto.nomeAttributo;
```

L'accesso al metodo di un oggetto provoca l'esecuzione del relativo blocco di codice. Inoltre in quest'esempio abbiamo definito una variabile intera `risultato` che ha immagazzinato il risultato della somma. Se non avessimo fatto ciò, non avrebbe avuto senso definire un metodo che restituisse un valore dal momento che non l'avremmo utilizzato in qualche modo! Da notare che avremmo anche potuto aggiungere alla riga 6:

```
int risultato = oggetto1.somma(5, 6);
```

la riga seguente:

```
System.out.println(risultato);
```

o equivalentemente sostituire la riga 6 con:

```
System.out.println(oggetto1.somma(5, 6));
```

In questo modo, stampando a video il risultato, avremmo potuto verificare in fase di runtime del programma la reale esecuzione della somma. Esistono anche metodi che non hanno parametri in input. Per esempio un metodo che somma sempre gli stessi numeri e quindi restituisce sempre lo stesso valore come il seguente:

```
public class AritmeticaFissa {  
    public int somma() {  
        return (5 + 6);  
    }  
}
```

oppure metodi che non hanno tipo di ritorno e quindi dichiarano come tipo di ritorno `void` (vuoto) come il metodo `main()`. Infatti il metodo `main()` è il punto di partenza del runtime di un'applicazione Java, quindi non deve restituire un risultato, non essendo chiamato esplicitamente da un altro metodo.

Esistono anche metodi che non hanno né parametri in input né in output (tipo di ritorno `void`), come per esempio un semplice metodo che visualizza a video sempre lo stesso messaggio:

```
public class Saluti {
```

```
public void stampaSaluto() {  
    System.out.println("Ciao");  
}
```

Segue una banale classe che istanzia un oggetto dalla classe `Saluti` e chiama il metodo `stampaSaluto()`. Come si può notare, se il metodo ha come tipo di ritorno `void`, non bisogna “catturarne” il risultato.

```
public class Due {  
    public static void main(String args[]) {  
        Saluti oggetto1 = new Saluti();  
        oggetto1.stampaSaluto();  
    }  
}
```

2.3.3 Varargs

È possibile anche utilizzare metodi che dichiarano come lista di parametri i cosiddetti **variable arguments**, più brevemente noti come **varargs**. In pratica è possibile creare metodi che dichiarano un numero non definito di parametri di un certo tipo. La sintassi è un po' particolare, e fa uso di tre puntini (come i puntini sospensivi), dopo la dichiarazione del tipo. Per esempio, è possibile invocare il seguente metodo:

```
public class AritmeticaVariabile {  
    public void somma(int... interi) {  
        //codice complicato. . .  
    }  
}
```

in uno qualsiasi dei seguenti modi:

```
AritmeticaVariabile ogg = new AritmeticaVariabile();  
ogg.somma();  
ogg.somma(1,2);  
ogg.somma(1,4,40,27,48,27,36,23,45,67,9,54,66,43);  
ogg.somma(1);
```

Quindi possiamo passare da 0 a n argomenti. Si noti che per ogni dichiarazione di metodo, è possibile dichiarare un solo parametro varargs come ultimo argomento. Quindi le seguenti dichiarazioni non compilerebbero:

```
public void somma(int... interi, int altroIntero) {  
    // . . .  
}  
.  
.  
.  
public void somma(int... interi, int... altriIntero) {
```

```
// ...
}
```

2.4 Le variabili in Java

Nella programmazione tradizionale, una variabile è una porzione di memoria in cui è immagazzinato un certo tipo di dato. Per esempio, un intero in Java è immagazzinato in 32 bit. I tipi di Java saranno argomento del prossimo modulo.

Anche per l'utilizzo delle variabili possiamo distinguere due fasi: dichiarazione ed assegnazione. L'assegnazione di un valore ad una variabile è un'operazione che si può ripetere molte volte nell'ambito esecutivo, anche contemporaneamente alla dichiarazione stessa.

2.4.1 Dichiarazione di una variabile

La sintassi per la dichiarazione di una variabile è la seguente:

```
[modificatori] tipo_di_dato nome_della_variabile [ = inizializzazione];
```

dove:

- ❑ **modificatori:** parole chiavi di Java che possono essere usate per modificare in qualche modo le funzionalità e le caratteristiche della variabile;
- ❑ **tipo di dato:** il tipo di dato della variabile;
- ❑ **nome della variabile:** identificatore della variabile;
- ❑ **inizializzazione:** valore di default con cui è possibile valorizzare una variabile.

Segue una classe che definisce in maniera superficiale un quadrato:

```
public class Quadrato {
    public int altezza;
    public int larghezza;
    public final int NUMERO_LATI = 4;
}
```

In questo caso abbiamo definito due variabili intere chiamate `altezza` e `larghezza`, ed una costante, `NUMERO_LATI`. Il modificatore `final` infatti (che sarà trattato in dettaglio nel modulo 6), rende una variabile costante nel suo valore. È anche possibile definire due variabili dello stesso tipo con un'unica istruzione come nel seguente esempio:

```
public class Quadrato {
    public int altezza, larghezza;
    public final int NUMERO_LATI = 4;
}
```

In questo caso possiamo notare una maggiore compattezza del codice ottenuta evitando le

duplicazioni, ma una minore leggibilità.

La dichiarazione di una variabile è molto importante. La posizione della dichiarazione definisce lo **scope**, ovvero la visibilità, ed il ciclo di vita della variabile stessa. In pratica potremmo definire tre diverse tipologie di variabili, basandoci sul posizionamento della dichiarazione:

2.4.2 Variabili d'istanza

Una variabile è detta variabile d'istanza (o anche attributo) se è dichiarata in una classe, ma al di fuori di un metodo.

Non è possibile comunque definire una variabile al di fuori di una classe.

Le variabili definite nella classe `Quadrato` sono tutte di istanza. Esse condividono il proprio ciclo di vita con l'oggetto (istanza) cui appartengono. Quando un oggetto della classe `Quadrato` è istanziato, viene allocato spazio per tutte le sue variabili d'istanza che vengono inizializzate ai relativi valori nulli (una tabella esplicativa è presentata nel modulo successivo dove verranno spiegati tutti i tipi di dati). Nel nostro caso, le variabili `altezza` e `larghezza` saranno inizializzate a zero. `NUMERO_LATI` è una costante esplicitamente inizializzata a 4, ed il suo valore non potrà cambiare. Una variabile d'istanza smetterà di esistere quando smetterà di esistere l'oggetto a cui appartiene.

2.4.3 Variabili locali

Una variabile è detta **locale** (o **di stack**, o **automatica**, o anche **temporanea**) se è dichiarata all'interno di un metodo. Essa smetterà di esistere quando terminerà il metodo. Una variabile di questo tipo, a differenza di una variabile di istanza, non sarà inizializzata ad un valore nullo al momento dell'istanza dell'oggetto a cui appartiene. È buona prassi inizializzare comunque una variabile locale ad un valore di default, nel momento in cui la si dichiara. Infatti il compilatore potrebbe restituire un messaggio di errore laddove ci sia possibilità che la variabile non venga inizializzata al runtime. Nel seguente esempio la variabile `z` è una variabile locale:

```
public int somma(int x, int y) {  
    int z = x + y;  
    return z;  
}
```

2.4.4 Parametri formali

Le variabili dichiarate all'interno delle parentesi tonde che si trovano alla destra dell'identificatore nella dichiarazione di un metodo, sono dette **parametri** o **argomenti** del metodo.

Per esempio, nella seguente dichiarazione del metodo `somma()` vengono dichiarati due parametri interi `x` e `y`:

```
public int somma(int x, int y) {
```

```
        return (x + y);  
    }
```

I parametri di un metodo saranno inizializzati, come abbiamo già potuto notare, al momento della chiamata del metodo. Infatti, per chiamare il metodo `somma()` dovremo passare i valori ai parametri, per esempio come nel seguente esempio:

```
int risultato = oggetto1.somma(5, 6);
```

In particolare, all'interno del metodo `somma()`, la variabile `x` varrà 5, mentre `y` varrà 6. Dal momento che il passaggio di parametri avviene sempre per valore, potremmo anche scrivere:

```
int a = 5, b = 6;  
int risultato = oggetto1.somma(a, b);
```

ed ottenere lo stesso risultato.

È importante sottolineare che un parametro si può considerare anche una variabile locale del metodo, avendo stessa visibilità e ciclo di vita. Le differenze sussistono solo nella posizione della dichiarazione, non nel processo di immagazzinamento in memoria.

Il concetto e la modalità di allocazione di memoria di una variabile d'istanza si differenziano a tal punto dai concetti e dalle modalità di allocazione di memoria di una variabile locale (o di un parametro) che è possibile assegnare in una stessa classe ad una variabile locale (o un parametro) e ad una variabile d'istanza lo stesso identificatore (nome).

Condividendo il ciclo di vita con il metodo in cui sono dichiarate, non ha senso (e non è possibile) anteporre alle variabili locali un modificatore d'accesso come `public`.

2.5 I metodi costruttori

In Java esistono metodi speciali che hanno “proprietà”. Tra questi c'è da considerare il metodo costruttore, che possiede le seguenti caratteristiche:

1. Ha lo stesso nome della classe.
2. Non ha tipo di ritorno.
3. È chiamato automaticamente (e solamente) ogni volta che è istanziato un oggetto, relativamente a quell'oggetto.
4. È presente in ogni classe.
5. Solitamente un metodo costruttore viene definito allo scopo di inizializzare le variabili d'istanza.

2.5.1 Caratteristiche di un costruttore

Un costruttore ha sempre e comunque lo stesso nome della classe in cui è dichiarato. È importante anche fare attenzione a lettere maiuscole e minuscole. Il fatto che non abbia tipo di ritorno non significa che il tipo di ritorno è `void`, ma che non dichiara alcun tipo di ritorno!

Per esempio, viene presentata una classe con un costruttore dichiarato esplicitamente:

```
public class Punto {  
    public Punto() //metodo costruttore {  
        System.out.println("Costruito un Punto!");  
    }  
    int x;  
    int y;  
}
```

Si noti che verrà eseguito il blocco di codice del costruttore ogni volta che sarà istanziato un oggetto. Analizziamo meglio la sintassi che permette di istanziare oggetti. Per esempio:

```
Punto punto1; //dichiarazione  
punto1 = new Punto(); // istanza
```

che è equivalente a:

```
Punto punto1 = new Punto(); //dichiarazione ed istanza
```

Come accennato in precedenza, è la parola chiave `new` che istanzia formalmente l'oggetto. Perciò basterebbe la sintassi:

```
new Punto();
```

per istanziare l'oggetto. In questo modo però, l'oggetto appena creato non avrebbe un nome (si dirà *reference* o *riferimento*) e quindi non sarebbe utilizzabile. Solitamente quindi, quando si istanzia un oggetto, gli si assegna un reference dichiarato precedentemente.

Appare quindi evidente che l'ultima parte dell'istruzione da analizzare (`Punto()`) non va interpretata come “nomeDellaClasse con parentesi tonde”, bensì come “chiamata al metodo costruttore”. In corrispondenza dell'istruzione suddetta, un programma produrrebbe il seguente output:

```
Costruito un Punto!
```

Questo è l'unico modo per chiamare un costruttore. Un costruttore essendo definito senza un di tipo di ritorno non può considerarsi un metodo ordinario.

L'utilità del costruttore non è esplicita nell'esempio appena proposto. Essendo un metodo (anche se speciale) può avere una lista di parametri. Di solito un costruttore è utilizzato per inizializzare le variabili d'istanza di un oggetto. È quindi possibile codificare il seguente costruttore all'interno della

classe Punto:

```
public class Punto {  
    public Punto(int a, int b) {  
        x = a;  
        y = b;  
    }  
    public int x;  
    public int y;  
}
```

Con questa classe non sarà più possibile istanziare gli oggetti con la solita sintassi:

```
Punto punto1 = new Punto();
```

otterremmo un errore di compilazione, dal momento che staremmo cercando di chiamare un costruttore che non esiste (quello senza parametri)! Questo problema è dovuto alla mancanza del costruttore di default, come vedremo nel prossimo paragrafo.

La sintassi da utilizzare però, potrebbe essere la seguente.

```
Punto punto1 = new Punto(5,6);
```

Questa ci permetterebbe anche d'inizializzare l'oggetto direttamente senza essere costretti ad utilizzare l'operatore dot. Infatti la precedente riga di codice è equivalente alle seguenti:

```
Punto punto1 = new Punto();  
punto1.x = 5;  
punto1.y = 6;
```

È anche possibile fornire più costruttori all'interno della stessa classe, purché la lista degli argomenti in input differisca. Per esempio sarà possibile fornire entrambi i costruttori che abbiamo visto alla classe Punto:

```
public class Punto {  
    public Punto(){  
        // Questo costruttore non fa niente  
    }  
  
    public Punto(int a, int b) {  
        x = a;  
        y = b;  
    }  
    public int x;  
    public int y;  
}
```

In questo modo potremo istanziare oggetti dalla classe Punto in due modi, o usando il primo

costruttore:

```
Punto p = new Punto();
```

oppure usando il secondo:

```
Punto p = new Punto(12,14);
```

La possibilità di creare più costruttori nella stessa classe è una manifestazione del polimorfismo (trattato in dettaglio nel modulo 7) che si chiama “overload”. Questo permette di scrivere più metodi con lo stesso nome (ma con differente lista dei parametri). Ci sono altre regole da studiare ma le approfondiremo a tempo debito.

2.5.2 Costruttore di default

Quando creiamo un oggetto, dopo l’istanza che avviene grazie alla parola chiave `new`, c’è sempre una chiamata ad un costruttore. Il lettore potrà obiettare che nelle classi utilizzate fino a questo punto non abbiamo mai fornito costruttori. Eppure, come appena detto, abbiamo chiamato costruttori ogni volta che abbiamo istanziato oggetti! Java, linguaggio ideato da programmatore per programmatore, ha una caratteristica molto importante che molti ignorano. Spesso vengono inseriti automaticamente e in modo trasparente dal compilatore Java comandi non inseriti dal programmatore. Se infatti proviamo a compilare una classe sprovvista di costruttore, il compilatore ne fornisce uno implicitamente. Il costruttore inserito non contiene comandi che provocano qualche conseguenza visibile al programmatore. Esso è detto “costruttore di default” e non ha parametri. Ci giustifica il fatto che fino ad ora non abbiamo mai istanziato oggetti passando parametri al costruttore.

Se per esempio codificassimo la classe `Punto` nel modo seguente:

```
public class Punto {  
    public int x;  
    public int y;  
}
```

Al momento della compilazione il compilatore aggiungerebbe ad essa il costruttore di default:

```
public class Punto {  
    public Punto() {  
        //nel costruttore di default  
        //sembra non ci sia scritto niente . . .  
    }  
    public int x;  
    public int y;  
}
```

Ecco perché fino ad ora abbiamo istanziato gli oggetti di tipo `Punto` con la sintassi:

```
Punto p = new Punto();
```

anche senza la presenza esplicita di costruttori. Se non esistesse il costruttore di default, avremmo dovuto imparare prima i costruttori e poi gli oggetti.

Questa è una delle caratteristiche che fa sì che Java possa essere definito linguaggio semplice (più precisamente ne esalta la facilità di sviluppo)! Infatti, il fatto che sia inserito implicitamente dal compilatore Java un costruttore all'interno delle classi, ci ha permesso di parlare di istanze di oggetti senza per forza dover prima spiegare un concetto tanto singolare come il costruttore.

Sottolineiamo una volta di più che il costruttore di default viene inserito in una classe dal compilatore se e solo se il programmatore non ne ha fornito uno esplicitamente. Nel momento in cui il programmatore fornisce ad una classe un costruttore, sia esso con o senza parametri, il compilatore **non** inserirà quello di default. L'argomento sarà ulteriormente approfondito nel Modulo 6, dove scopriremo altre “istruzioni fantasma”.

L'ordine dei membri all'interno della classe non è importante. Possiamo scrivere prima i metodi, poi i costruttori e poi le variabili d'istanza, o alternare un costruttore una variabile e un metodo; non ha importanza per il compilatore. La creazione di un oggetto di una classe in memoria, infatti, non provoca l'esecuzione in sequenza del codice della classe come se fosse un programma funzionale. Solitamente però la maggior parte del codice che abbiamo visto tende a definire prima le variabili d'istanza, poi i costruttori ed infine i metodi.

Un costruttore senza parametri inserito dal programmatore non si chiama costruttore di default, è solo un costruttore senza parametri! La terminologia è importante, non dobbiamo confondere due concetti diversi tra loro.

2.6 Package

Un `package` in Java permette di raggruppare in un'unica entità complessa classi Java logicamente correlate. Fisicamente il package non è altro che una cartella del nostro sistema operativo, ma non tutte le cartelle sono package. Per eleggere una cartella a package, una classe Java deve dichiarare nel suo codice la sua appartenenza a quel determinato package, e inoltre deve risiedere fisicamente all'interno di essa.

Per esempio se volessimo far appartenere la classe `Punto` a un package `geometria`, dovremmo semplicemente inserire come prima istruzione del nostro file sorgente la seguente dichiarazione di package:

```
package geometria;  
  
public class Punto {  
    . . .
```

Questo comporterebbe che il file compilato (.class) dovrà risiedere in una cartella chiamata proprio “geometria”. Se invece volessimo creare un albero di package più strutturato possiamo tranquillamente usare la sintassi sfruttando l’operatore dot (che anche in questo caso significa *appartenenza*):

```
package studio.matematica.geometria;  
  
public class Punto {  
    . . .
```

In questo caso il file compilato dovrebbe risiedere all’interno di una cartella `geometria` a sua volta contenuta in una cartella `matematica` a sua volta contenuta in una cartella `studio`. Il problema con i package sono poi le regole di visibilità delle classi contenute in esse. Per esempio come fa una classe che si trova in un’altra cartella a “vedere” una classe che si trova in un’altra cartella? Sicuramente la classe che vuole usare la classe `Punto` (supponiamo si chiami `TestPunto`) se non si trova nello stesso package `studio.matematica.geometria` deve importare la classe `Punto` con il comando `import`, che va dichiarato dopo la dichiarazione di package ma prima della dichiarazione della classe, come nel seguente esempio:

```
package studio.matematica.geometria.test;  
import studio.matematica.geometria.Punto;  
  
public class TestPunto {  
    public static void main (String args[]) {  
        Punto p = new Punto();  
        . . .  
    }  
}
```

Purtroppo però la situazione non è così semplice come sembra. Senza un IDE che gestisce automaticamente il posizionamento dei file .java e dei file .class, ci sono diversi problemi da gestire. Giacché il concetto di package non è essenziale al fine di approcciare correttamente al linguaggio, è stato scelto di rimandare al modulo 5 la trattazione dettagliata di questo argomento. Fino a quel momento non useremo più i package nei nostri esempi e nei nostri esercizi.

Riepilogo

In questo modulo abbiamo introdotto i principali componenti di un’applicazione Java. Ogni file che fa parte di un’applicazione Java conterrà il listato di una ed una sola classe (tranne eccezioni di cui il lettore per adesso farà a meno). Inoltre, ogni **classe** è opzionalmente dichiarata appartenente ad un **package**. Ogni classe solitamente contiene definizioni di **variabili**, **metodi** e **costruttori**. Le **variabili** si dividono in tre tipologie, definite dal posizionamento della dichiarazione: **d’istanza**, **locali** e **parametri** (che si possono in sostanza considerare locali). Le variabili hanno un **tipo** (nel prossimo modulo tratteremo in dettaglio l’argomento) e un **valore**. In particolare le variabili d’istanza

rappresentano gli **attributi** (le caratteristiche) di un **oggetto**, ed è possibile accedervi tramite l'operatore **dot** applicato ad un oggetto. I **metodi** contengono codice operativo e definiscono le funzionalità degli oggetti. Hanno bisogno di essere dichiarati all'interno di una classe e di essere utilizzati tramite l'operatore dot applicato ad un oggetto. I metodi possono o meno restituire un risultato. I **costruttori** sono metodi speciali che si trovano all'interno di ogni classe. Infatti, se il programmatore non ne fornisce uno esplicitamente, il compilatore Java aggiungerà il **costruttore di default**. Abbiamo infine solo introdotto il concetto di **package** in quanto è anch'esso un componente fondamentale della programmazione Java, ma ne abbiamo rimandato la spiegazione dettagliata a quando avremo più elementi per poterne apprezzare l'utilizzo. Per ora, progettare un'applicazione anche semplice sembra ancora un'impresa ardua...

Esercizi modulo 2

Esercizio 2.a)

Viene fornita (copiare, salvare e compilare) la seguente classe:

```
public class NumeroIntero {  
    public int numeroIntero;  
    public void stampaNumero() {  
        System.out.println(numeroIntero);  
    }  
}
```

Questa classe definisce il concetto di numero intero come oggetto. In essa vengono dichiarati una variabile intera ed un metodo che stamperà la variabile stessa.

Scrivere, compilare ed eseguire una classe che:

- istanzierà almeno due oggetti dalla classe `NumeroIntero` (contenente ovviamente un metodo `main()`);
- cambierà il valore delle relative variabili e testerà la veridicità delle avvenute assegnazioni, sfruttando il metodo `stampaNumero()`;
- aggiungerà un costruttore alla classe `NumeroIntero` che inizializzi la variabile d'istanza.

Due domande ancora:

1. A che tipologia di variabili appartiene la variabile `numeroIntero` definita nella classe `NumeroIntero`?
2. Se istanziamo un oggetto della classe `NumeroIntero`, senza assegnare un nuovo valore alla variabile `numeroIntero`, quanto varrà quest'ultima?

Esercizio 2.b)

Concetti sui componenti fondamentali, Vero o Falso:

1. Una variabile d'istanza deve essere per forza inizializzata dal programmatore.

2. Una variabile locale condivide il ciclo di vita con l'oggetto in cui è definita.
3. Un parametro ha un ciclo di vita coincidente con il metodo in cui è dichiarato: nasce quando il metodo viene invocato, muore quando termina il metodo.
4. Una variabile d'istanza appartiene alla classe in cui è dichiarata.
5. Un metodo è sinonimo di azione, operazione.
6. Sia le variabili sia i metodi sono utilizzabili di solito mediante l'operatore dot, applicato ad un'istanza della classe dove sono stati dichiarati.
7. Un costruttore è un metodo che non restituisce mai niente, infatti ha come tipo di ritorno `void`.
8. Un costruttore viene detto di default, se non ha parametri.
9. Un costruttore è un metodo e quindi può essere utilizzato mediante l'operatore dot, applicato ad un'istanza della classe dove è stato dichiarato.
10. Un package è fisicamente una cartella che contiene classi, le quali hanno dichiarato esplicitamente di far parte del package stesso nei rispettivi file sorgente.

Esercizio 2.c)

Sintassi dei componenti fondamentali. Vero o Falso:

1. Nella dichiarazione di un metodo, il nome è sempre seguito dalle parentesi che circondano i parametri opzionali, ed è sempre preceduto da un tipo di ritorno.
2. Il seguente metodo è dichiarato in maniera corretta:

```
public void metodo () {  
    return 5;  
}
```

3. Il seguente metodo è dichiarato in maniera corretta:

```
public int metodo () {  
    System.out.println("Ciao");  
}
```

4. La seguente variabile è dichiarata in maniera corretta:

```
public int a = 0;
```

5. La seguente variabile `x` è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):

```
Punto p1 = new Punto();  
Punto.x = 10;
```

6. La seguente variabile `x` è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):

```
Punto p1 = new Punto();
Punto.p1.x = 10;
```

7. La seguente variabile `x` è utilizzata in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):

```
Punto p1 = new Punto();
x = 10;
```

8. Il seguente costruttore è utilizzato in maniera corretta (fare riferimento alla classe `Punto` definita in questo modulo):

```
Punto p1 = new Punto();
p1.Punto();
```

9. Il seguente costruttore è dichiarato in maniera corretta:

```
public class Computer {
    public void Computer() {
    }
}
```

10. Il seguente costruttore è dichiarato in maniera corretta:

```
public class Computer {
    public computer(int a) {
    }
}
```

Soluzioni esercizi modulo 2

Esercizio 2.a)

Di seguito viene proposta una classe che aderisce ai requisiti richiesti:

```
public class ClasseRichiesta {
    public static void main (String args []) {
        NumeroIntero uno = new NumeroIntero();
        NumeroIntero due = new NumeroIntero();
        uno.numeroIntero = 1;
        due.numeroIntero = 2;
        uno.stampaNumero();
        due.stampaNumero();
    }
}
```

Inoltre un costruttore per la classe `NumeroIntero` potrebbe impostare l'unica variabile d'istanza `numeroIntero`:

```
public class NumeroIntero {
    public int numeroIntero;
    public NumeroIntero(int n) {
        numeroIntero = n;
```

```
    }
    public void stampaNumero() {
        System.out.println(numeroIntero);
    }
}
```

In tal caso però, per istanziare oggetti dalla classe `NumeroIntero`, non sarà più possibile utilizzare il costruttore di default (che non sarà più inserito dal compilatore). Quindi la seguente istruzione produrrebbe un errore in compilazione:

```
NumeroIntero uno = new NumeroIntero();
```

Bisogna invece creare oggetti passando al costruttore direttamente il valore della variabile da impostare, per esempio:

```
NumeroIntero uno = new NumeroIntero(1);
```

Risposte alle due domande:

1. Trattasi di una variabile d'istanza, perché dichiarata all'interno di una classe, al di fuori di metodi.
2. Il valore sarà zero, ovvero il valore nullo per una variabile intera. Infatti, quando si istanzia un oggetto, le variabili d'istanza vengono inizializzate ai valori nulli, se non esplicitamente inizializzate ad altri valori.

Esercizio 2.b)

Concetti sui componenti fondamentali. Vero o Falso:

1. **Falso**, una variabile locale deve essere per forza inizializzata dal programmatore.
2. **Falso**, una variabile d'istanza condivide il ciclo di vita con l'oggetto in cui è definita.
3. **Vero**.
4. **Falso**, una variabile d'istanza appartiene ad un oggetto istanziato dalla classe in cui è dichiarata.
5. **Vero**.
6. **Vero**.
7. **Falso**, un costruttore è un metodo che non restituisce mai niente, infatti non ha tipo di ritorno.
8. **Falso**, un costruttore viene detto di default, se viene inserito dal compilatore. Inoltre non ha parametri.
9. **Falso**, un costruttore è un metodo speciale che ha la caratteristica di essere invocato una ed una sola volta nel momento in cui si istanzia un oggetto.
10. **Vero**.

Esercizio 2.c)

Sintassi dei componenti fondamentali. Vero o Falso:

1. Vero.

2. Falso, tenta di restituire un valore intero avendo tipo di ritorno `void`.

3. Falso, il metodo dovrebbe restituire un valore intero.

4. Vero.

5. Falso, l'operatore **dot** deve essere applicato all'oggetto e non alla classe:

```
Punto p1 = new Punto();
```

```
p1.x = 10;
```

6. Falso, l'operatore **dot** deve essere applicato all'oggetto e non alla classe, ed inoltre la classe non "contiene" l'oggetto.

7. Falso, l'operatore **dot** deve essere applicato all'oggetto. Il compilatore non troverebbe infatti la dichiarazione della variabile `x`.

8. Falso, un costruttore è un metodo speciale che ha la caratteristica di essere invocato una ed una sola volta nel momento in cui si istanzia un oggetto.

9. Falso, il costruttore non dichiara tipo di ritorno e deve avere nome coincidente con la classe.

10. Falso, il costruttore deve avere nome coincidente con la classe.

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi

<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper definire i concetti di classe, oggetto, variabile, metodo e costruttore (unità 2.1, 2.2, 2.3, 2.4, 2.5)	<input type="checkbox"/>	
Saper dichiarare una classe (unità 2.1)	<input type="checkbox"/>	
Istantiare oggetti da una classe (unità 2.2)	<input type="checkbox"/>	
Utilizzare i membri pubblici di un oggetto sfruttando l'operatore dot (unità 2.2, 2.3, 2.4)	<input type="checkbox"/>	
Dichiarare ed invocare un metodo (unità 2.3)	<input type="checkbox"/>	
Saper dichiarare e inizializzare una variabile (unità 2.4)	<input type="checkbox"/>	
Saper definire ed utilizzare i diversi tipi di variabili (d'istanza, locali e parametri formali) (unità 2.4)	<input type="checkbox"/>	
Dichiarare ed invocare un metodo costruttore (unità 2.5)	<input type="checkbox"/>	
Saper definire un costruttore di default (unità 2.5)	<input type="checkbox"/>	

Note:

Identifieri, tipi di dati e array

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Saper utilizzare le convenzioni per il codice Java (unità 3.1).
- ✓ Conoscere e saper utilizzare tutti i tipi di dati primitivi (unità 3.2).
- ✓ Saper gestire casting e promotion (unità 3.2).
- ✓ Saper utilizzare i reference, e capirne la filosofia (unità 3.3).
- ✓ Iniziare ad esplorare la documentazione della libreria standard di Java (unità 3.4).
- ✓ Saper utilizzare la classe String (unità 3.4).
- ✓ Saper utilizzare gli array (unità 3.5).
- ✓ Saper commentare il proprio codice ed essere in grado di utilizzare lo strumento javadoc per produrre documentazione tecnica esterna (unità 3.1, 3.4).

In questo modulo saranno dapprima definite alcune regole fondamentali che la programmazione Java richiede. Poi passeremo ad introdurre i tipi di dati definiti dal linguaggio e tutte le relative problematiche. Per completare il discorso introdurremo gli array, che in Java sono oggetti abbastanza “particolari” rispetto ad altri linguaggi.

3.1 Stile di codifica

Il linguaggio Java:

- è a schema libero.
- È case sensitive.
- Supporta i commenti.
- Definisce parole chiave.
- Ha regole ferree per i tipi di dati, ed alcune semplici convenzioni per i nomi.

3.1.1 Schema libero

Potremmo scrivere un intero programma in Java tutto su di un'unica riga, oppure andando a capo dopo ogni parola: il compilatore non avrà problemi a compilare il codice se esso è corretto. Il problema semmai è dello sviluppatore che avrà difficoltà a capirne il significato!

Esistono quindi metodi standard d'indentazione del codice, che facilitano la lettura di un

programma Java. Di seguito è presentata una semplice classe che utilizza uno dei due più usati metodi di formattazione:

```
public class Classe
{
    public int intero;
    public void metodo()
    {
        intero = 10;
        int unAltroIntero = 11;
    }
}
```

Con questo stile (che è utilizzato anche dai programmati C), il lettore può capire subito dove la classe ha inizio e dove ha fine, dato che le parentesi graffe che delimitano un blocco di codice sono incolonnate. Stesso discorso per il metodo: è evidente dove inizia, dove finisce ed il codice che ne definisce il comportamento.

Ma lo stile più utilizzato dai programmati Java è il seguente:

```
public class Classe {
    public int intero;
    public void metodo() {
        intero = 10;
        int unAltroIntero = 11;
    }
}
```

per il quale valgono circa le stesse osservazioni fatte per il primo metodo.

Si raccomanda, per una buona riuscita del lavoro che sarà svolto in seguito dal lettore, una rigorosa applicazione di uno dei due stili appena presentati. In questo manuale utilizzeremo entrambi gli stili.

Se utilizzate EJE come editor, potete formattare il vostro codice con entrambi gli stili, mediante la pressione del menu apposito, o tramite il relativo pulsante sulla barra degli strumenti, o attraverso la scorciatoia di tastiera CTRL-SHIFTF. Per configurare lo stile da utilizzare, scegliere il menu File ➔ Options (o premere F12) e impostare il “Braces style” con lo stile desiderato, nel tab “Editor”.

Alcuni dei tipici errori che il programmatore alle prime armi commette sono semplici dimenticanze. È frequente dimenticare di chiudere una parentesi graffa di una classe, o dimenticare il ";" dopo un'istruzione, o le parentesi tonde che seguono l'invocazione di un metodo. Per questo è consigliato al lettore di abituarsi a scrivere ogni istruzione in maniera completa, per poi pensare a formattare in maniera corretta. Per esempio, se dichiariamo una classe è buona prassi scrivere entrambe le parentesi graffe, prima di scrivere il codice contenuto in esse. I tre passi seguenti dovrebbero

chiarire il concetto al lettore:

- passo 1: dichiarazione:

```
public class Classe {}
```

- passo 2: formattazione:

```
public class Classe {  
}
```

- passo 3: completamento:

```
public class Classe {  
    public int intero;  
    ...  
}
```

3.1.2 Case sensitive

Java è un linguaggio case sensitive, ovvero fa distinzione tra lettere maiuscole e minuscole. Il programmatore alle prime armi tende a digerire poco questa caratteristica del linguaggio. Bisogna ricordarsi di non scrivere ad esempio `class` con lettera maiuscola, perché per il compilatore non significa niente. L'identificatore `unAltroIntero` è diverso dall'identificatore `unalternoIntero`. Bisogna quindi fare attenzione e, specialmente nei primi tempi, avere un po' di pazienza.

3.1.3 Commenti

Commentare opportunamente il codice è una pratica che dovrebbe essere considerata obbligatoria dal programmatore che inizia a programmare. I commenti non solo chiariscono il codice, ma anche obbligano a capire bene cosa si sta scrivendo visto che bisogna descriverlo.

Java supporta tre modi diversi per commentare il codice:

1. Commenti su un'unica riga:

```
// Questo è un commento su una sola riga
```

2. Commento su più righe:

```
/*  
Questo è un commento  
su più righe  
*/
```

3. Commento Javadoc:

```
/**  
Questo commento permette di produrre  
la documentazione del codice
```

```
in formato HTML, nello standard Javadoc  
*/
```

Nel primo caso tutto ciò che scriveremo su di una riga dopo aver scritto “//” non sarà preso in considerazione dal compilatore. Questa sintassi permetterà di commentare brevemente alcune parti di codice. Per esempio:

```
// Questo è un metodo  
public void metodo() {  
    . . .  
}  
  
public int a; //Questa è una variabile
```

Il commento su più righe potrebbe essere utile ad esempio per la descrizione delle funzionalità di un programma. In pratica il compilatore non prende in considerazione tutto ciò che scriviamo tra “/*” ed “*/”. Per esempio:

```
/*  
Questo metodo è commentato...  
public void metodo() {  
    . . .  
}  
*/
```

Queste due prime tipologie di commenti sono ereditate dal linguaggio C++.

Ciò che invece rappresenta una novità è il terzo tipo di commento. L'utilizzo in pratica è lo stesso del secondo tipo, e permette di definire commenti su più righe. In più offre la possibilità di produrre in modo standard, in formato HTML, la documentazione tecnica del programma, sfruttando un comando chiamato `javadoc`. Per esempi pratici d'utilizzo il lettore è rimandato alla fine di questo modulo.

3.1.4 Regole per gli identificatori

Gli identificatori (nomi) dei metodi, delle classi, degli oggetti, delle variabili e delle costanti (ma anche delle interfacce, delle enumerazioni e delle annotazioni che studieremo più avanti) hanno due regole da rispettare.

1. Un identificatore non può coincidere con una parola chiave (keyword) di Java. Infatti una parola chiave ha un certo significato per il linguaggio di programmazione. Tutte le parole chiave di Java sono costituite da lettere minuscole. Nella seguente tabella sono riportate tutte le parole chiave di Java in ordine alfabetico:

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while	enum	@interface

Possiamo notare alcune parole chiave che abbiamo già incontrato come `int`, `public`, `void`, `return` e `class`, e di cui già conosciamo il significato. Quindi non potremo chiamare una variabile `class`, oppure un metodo `void`.

Possiamo notare anche le parole riservate `goto` e `const`, che non hanno nessun significato in Java, ma che non possono essere utilizzate come identificatori. Esse sono dette parole riservate (reserved words). Si noti anche che `@interface` (parola utilizzata per dichiarare le annotazioni, di cui parleremo più avanti) iniziando con il simbolo di “chiocciola” `@`, non potrebbe comunque essere utilizzata quale identificatore, come spiegato nel prossimo punto. Anche in questo caso quindi non si dovrebbe parlare tecnicamente di parola chiave.

2. In un identificatore:

- il primo carattere può essere A-Z, a-z, _, \$
- il secondo ed i successivi possono essere A-Z, a-z, _, \$, 0-9

Quindi: “`a2`” è un identificatore valido, mentre “`2a`” non lo è.

3.1.5 Standard e convenzioni per i nomi

Se definiamo gli identificatori (nomi) rispettando le due regole appena descritte, non otterremo errori in compilazione. Tuttavia esistono direttive fornite direttamente da Oracle (e originariamente da Sun Microsystems) per raggiungere uno standard anche nello stile d’implementazione. È importantissimo utilizzare queste direttive in un linguaggio tanto standardizzato quanto Java.

1. Gli identificatori devono essere significativi. Infatti, se scrivessimo un programma utilizzando la classe `a`, che definisce le variabili `b`, `c`, `d` e il metodo `e`, sicuramente ridurremmo la comprensibilità del programma stesso.
2. Di solito l’identificatore di una variabile è composto da uno o più sostantivi, per esempio `numeroLatini`, `O larghezza` o anche `numeroPartecipantiAlSimposio`. Gli identificatori dei

metodi solitamente conterranno verbi, ad esempio `stampaNumeroPartecipantiAlSimposio`, o somma.

3. Esistono convenzioni per gli identificatori, così come in molti altri linguaggi. In Java sono semplicissime:

Convenzione per le classi:

Un identificatore di una classe (ma questa regola vale anche per le interfacce, le enumerazioni e le annotazioni che studieremo più avanti) deve sempre iniziare con una lettera maiuscola. Se composto da più parole, queste non si possono separare, perché il compilatore non può intuire le nostre intenzioni. Come abbiamo notato in precedenza, bisogna invece unire le parole in modo tale da formare un unico identificatore, e fare iniziare ognuna di esse con lettera maiuscola. Esempi di identificatori per una classe potrebbero essere:

- Persona
- MacchinaDaCorsa
- FiguraGeometrica

Convenzione per le variabili:

Un identificatore di una variabile deve sempre iniziare per lettera minuscola. Se l'identificatore di una variabile deve essere composto da più parole, valgono le stesse regole in uso per gli identificatori delle classi (tranne il fatto che la prima lettera deve sempre essere minuscola). Quindi esempi di identificatori per una variabile potrebbero essere:

- pesoSpecifico
- numeroDiMinutiComplessivi
- x

Convenzione per i metodi:

Per un identificatore di un metodo valgono le stesse regole in uso per gli identificatori delle variabili. Potremo in ogni caso sempre distinguere un identificatore di una variabile da un identificatore di un metodo, giacché quest'ultimo è sempre seguito da parentesi tonde. Inoltre, per quanto già affermato, il nome di un metodo dovrebbe contenere almeno un verbo. Quindi, esempi di identificatori validi per un metodo potrebbero essere:

- sommaDueNumeri(int a, int b)
- cercaUnaParola(String parola)
- stampa()

Convenzione per le costanti:

Gli identificatori delle costanti invece, si devono distinguere nettamente dagli altri, e tutte le lettere dovranno essere maiuscole. Se l'identificatore è composto di più parole, queste si separano con un underscore (simbolo di sottolineatura). Per esempio:

- NUMERO_LATI_DI_UN_QUADRATO
- PI_GRECO

Non esiste un limite al numero dei caratteri di cui può essere composto un identificatore.

3.2 Tipi di dati primitivi

Java definisce solamente otto tipi di dati primitivi:

- Tipi interi: byte, short, int, long.
- Tipi floating point (o a virgola mobile): float e double.
- Tipo testuale: char.
- Tipo logico-boleano: boolean.

In Java non esistono tipi senza segno (unsigned), e per la rappresentazione dei numeri interi negativi viene utilizzata la regola del “complemento a due”. Eviteremo di scendere in questi dettagli perché li consideriamo poco utili ai fini della conoscenza di Java.

3.2.1 Tipi di dati interi, casting e promotion

I tipi di dati interi sono quattro: byte, short, int e long. Essi condividono la stessa funzionalità (tutti possono immagazzinare numeri interi positivi o negativi) ma differiscono per quanto riguarda il proprio intervallo di rappresentazione. Infatti un byte può immagazzinare un intero utilizzando un byte (otto bit), uno short due byte (16 bit), un int quattro byte (32 bit) ed un long otto byte (64 bit). Lo schema seguente riassume dettagliatamente i vari intervalli di rappresentazione:

Tipo	Intervallo di rappresentazione
byte	8 bit (da -128 a +127)
short	16 bit (da -32.768 a +32.767)
int	32 bit (da -2.147.483.648 a +2.147.483.647)
long	64 bit (da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807)

Si noti che l'intervallo di rappresentazione dei tipi primitivi interi è sempre compreso tra un minimo di -2 elevato al numero di bit meno uno, ed un massimo di 2 elevato al numero di bit meno uno, sottraendo alla fine un'unità. Quindi potremmo riscrivere la tabella precedente nel seguente modo:

Tipo	Intervallo di rappresentazione
byte	8 bit (da -2^7 a 2^7-1)
short	16 bit (da -2^{15} a $2^{15}-1$)

int	32 bit (da -2 ³¹ a 2 ³¹ -1)
long	64 bit (da -2 ⁶³ a 2 ⁶³ -1)

Si noti anche che per ogni tipo numerico esiste un numero pari di numeri positivi e di numeri negativi. Infatti il numero 0 (zero) è considerato un numero positivo.

Per immagazzinare un intero si possono utilizzare quattro forme:

1. Decimale (o notazione naturale): quella che usiamo abitualmente ogni giorno.
2. Binaria: la notazione che utilizzano i processori dei computer, composta solo da 0 e 1. Con 8 bit (un byte) come detto, è possibile rappresentare tutti i numeri che vanno da -128 a +127.
3. Ottale: si utilizzano solo i numeri da 0 a 7.
4. Esadecimale: si utilizzano oltre ai numeri da 0 a 9 anche le lettere da A ad F.

Per la notazione binaria bisogna anteporre al numero intero 0 (zero) e una b (maiuscola o minuscola). Per la notazione ottale basta anteporre al numero intero uno 0 (zero). Per la notazione esadecimale 0 e x (indifferentemente maiuscola o minuscola). Per la notazione decimale (o naturale) non c'è bisogno di utilizzare prefissi. Segue qualche esempio d'utilizzo di tipi interi:

```
byte b = 10; //notazione decimale: b vale 10
short s = 022; //notazione ottale: s vale 18
long l = 0x12acd; //notazione esadecimale: l vale 76493
int i = 1000000000; //notazione decimale: i vale 1000000000
int n = 0b10100001010001011010000101000101 //notazione binaria:
                                                //n vale -1589272251
```

Ci sono alcune osservazioni da fare riguardo i tipi di dati primitivi. Facciamo subito un esempio, consideriamo la seguente assegnazione:

```
byte b = 127;
```

Il compilatore è in grado di capire che 127 è un numero appartenente all'intervallo di rappresentazione dei byte e quindi l'espressione precedente è corretta e compilabile senza problemi. Al contrario le seguenti istruzioni:

```
byte b = 128; //il massimo per byte è 127
short s = 32768; //il massimo per short è 32767
int i = 2147483648; //il massimo per int è 2147483647
```

provocano errori in compilazione.

C'è da fare però una precisazione. Consideriamo la seguente istruzione:

```
byte b = 50;
```

Questo statement è corretto. Il numero intero 50 è tranquillamente compreso nell'intervallo di rappresentazione di un `byte`, che va da -128 a +127. Il compilatore determina la grandezza del valore numerico, e controlla se è compatibile con il tipo di dato dichiarato. Allora consideriamo quest'altro statement:

```
b = b * 2;
```

Ciò darà luogo ad un errore in compilazione! Infatti il compilatore non eseguirà l'operazione di moltiplicazione per controllare la compatibilità con il tipo di dato dichiarato. Invece promuoverà automaticamente i due operandi ad `int`. Quindi, se `50*2` è un `int`, non può essere immagazzinato in `b` che è un `byte`.

Il fenomeno appena descritto è conosciuto come “promozione automatica nelle espressioni” (promotion). Per gli operatori binari esistono quattro regole, che dipendono dai tipi degli operandi in questione:

- se uno degli operandi è `double`, l'altro operando sarà convertito in `double`;
- se il più ampio degli operandi è un `float`, l'altro operando sarà convertito in `float`;
- se il più ampio degli operandi è un `long`, l'altro operando sarà convertito in `long`;
- in ogni altro caso entrambi gli operandi saranno convertiti in `int`.

La promozione automatica ad intero degli operandi avviene prima che sia eseguita una qualsiasi operazione binaria.

Ma, evidentemente, `50*2` è immagazzinabile in un `byte`. Esiste una tecnica per forzare una certa quantità ad essere immagazzinata in un certo tipo di dato. Questa tecnica è nota come `cast` (o `casting`). La sintassi da utilizzare per risolvere il nostro problema è:

```
b = (byte)(b * 2);
```

In questo modo il compilatore sarà avvertito che un'eventuale perdita di precisione, è calcolata e sotto controllo. Bisogna essere però molto prudenti nell'utilizzare il casting in modo corretto. Infatti se scrivessimo:

```
b = (byte)128;
```

il compilatore non segnalerebbe nessun tipo d'errore. Siccome il cast agisce troncando i bit in eccedenza (nel nostro caso, dato che un `int` utilizza 32 bit, mentre un `byte` solo 8, saranno troncati i primi 24 bit dell'`int`), la variabile `b` avrà il valore di -128 e non di 128!

Un altro tipico problema di cui preoccuparsi è la somma di due interi. Per le stesse ragioni di cui sopra, se la somma di due interi supera il range consentito, è comunque possibile immagazzinarne il risultato in un intero senza avere errori in compilazione ma il risultato sarà diverso da quello previsto. Per esempio, le seguenti istruzioni saranno compilate senza errori:

```
int a = 2147483647; // Massimo valore per un int  
int b = 1;  
int risultato = a+b;
```

ma il valore della variabile risultato sarà di -2147483648!

Anche la divisione tra due interi rappresenta un punto critico! Infatti il risultato finale, per quanto detto sinora, non potrà che essere immagazzinato in un intero, ignorando così eventuali cifre decimali.

Inoltre, se utilizziamo una variabile `long`, a meno di cast esplicativi, essa sarà sempre inizializzata con un intero. Quindi, se scriviamo:

```
long l = 2000;
```

Dobbiamo tener ben presente che `2000` è un `int` per default, ma il compilatore non ci segnalera errori perché un `int` può essere tranquillamente immagazzinato in un `long`. Per la precisione dovremmo scrivere:

```
long l = 2000L;
```

che esegue con una sintassi più compatta il casting da `int` a `long`. Quindi, un cast a `long` si ottiene con una sintassi diversa dal solito, posponendo una “elle” maiuscola o minuscola al valore intero assegnato.

Si preferisce utilizzare la notazione maiuscola dato che una “elle” minuscola si può confondere con un numero “uno” in alcuni ambienti di sviluppo. Si noti che saremmo obbligati ad un casting a `long` nel caso in cui volessimo assegnare alla variabile `l` un valore fuori dell’intervallo di rappresentazione di un `int`. Per esempio:

```
long l = 3000000000;
```

produrrebbe un errore in compilazione. Bisogna eseguire il casting nel seguente modo:

```
long l = 3000000000L;
```

In generale comunque l’`int` è il tipo intero che viene usato più spesso.

3.2.2 Tipi di dati a virgola mobile, casting e promotion

Java utilizza per i valori floating point (a virgola mobile) lo standard di decodifica IEEE-754. I due tipi che possiamo utilizzare sono:

Operatore	Intervallo di rappresentazione
<code>float</code>	32 bit (da $+/-1.40239846^{-45}$ a $+/-3.40282347^{+38}$)
<code>double</code>	64 bit (da $+/-4.94065645841246544^{-324}$ a $+/-1.79769313486231570^{+328}$)

È anche possibile utilizzare la notazione esponenziale o ingegneristica (la “e” può essere sia maiuscola sia minuscola), per esempio:

```
double d = 1.26E-2; //equivalente a 1.26 diviso 100 = 0.0126
```

Per quanto riguarda cast e promotion, la situazione cambia rispetto al caso dei tipi interi. Il default è `double` e non `float` come ci si potrebbe aspettare. Ciò implica che, se vogliamo assegnare un valore a virgola mobile ad un `float`, non possiamo fare a meno di un cast. Per esempio, la seguente riga di codice provocherebbe un errore in compilazione:

```
float f = 3.14;
```

Anche in questo caso il linguaggio ci viene incontro permettendoci il cast con la sintassi breve:

```
float f = 3.14F;
```

La “effe” può essere sia maiuscola sia minuscola.

Esiste, per quanto ridondante, anche la forma contratta per i `double`:

```
double d = 10.12E24;
```

è equivalente a scrivere:

```
double d = 10.12E24D;
```

Alcune operazioni matematiche potrebbero dare risultati che non sono compresi nell’insieme dei numeri reali (per esempio “infinito”). Nella libreria standard sono definite delle classi dette **classi wrapper** (in italiano **classi involucro**), che non sono altro che classi che rappresentano i tipi di dati primitivi. Per esempio esiste la classe `Integer` per il tipo primitivo `int`, la classe `short` per il tipo di dato `short` e così via. Gli oggetti istanziati da queste classi sono interscambiabili con i dati primitivi grazie alla caratteristica di Java nota come **autoboxing-autounboxing**. È per questo che le classi wrapper `Double` e `Float` forniscono le seguenti costanti statiche:

- `Float.NaN`
- `Float.NEGATIVE_INFINITY`
- `Float.POSITIVE_INFINITY`
- `Double.NaN`
- `Double.NEGATIVE_INFINITY`
- `Double.POSITIVE_INFINITY`

Dove `NaN` sta per “Not a Number” (“non un numero”). Per esempio:

```
double d = -10.0 / 0.0;
System.out.println(d);
```

produrrà il seguente output:

NEGATIVE_INFINITY

Incontreremo le classe wrapper più volte durante la lettura di questo libro.

È bene aprire una piccola parentesi sui modificatori `final` e `static`.

Una costante statica in Java è una variabile dichiarata `final` e `static`. In particolare, il modificatore `final` applicato ad una variabile farà in modo che ogni tentativo di cambiare il valore di tale variabile (già inizializzata) produrrà un errore in compilazione. Quindi una variabile `final` è una costante.

Per quanto riguarda il modificatore `static`, come abbiamo già avuto modo di asserire precedentemente, il discorso è più complicato. Quando dichiariamo una variabile statica, tale variabile sarà condivisa da tutte le istanze di quella classe. Questo significa che, se abbiamo la seguente classe che dichiara una variabile statica (anche detta “variabile di classe”):

```
public class MiaClasse {  
    public static int variabileStatica = 0;  
}
```

se istanziamo oggetti da questa classe, essi condivideranno il valore di `variabileStatica`. Per esempio, il seguente codice:

```
MiaClasse ogg1 = new MiaClasse();  
MiaClasse ogg2 = new MiaClasse();  
System.out.println(ogg1.variabileStatica + "-"  
+ ogg2.variabileStatica);  
ogg1.variabileStatica = 1;  
System.out.println(ogg1.variabileStatica + "-"  
+ ogg2.variabileStatica);  
ogg2.variabileStatica = 2;  
System.out.println(ogg1.variabileStatica + "-"  
+ ogg2.variabileStatica);
```

produrrà il seguente output:

```
0-0  
1-1  
2-2
```

I modificatori `static` e `final`, saranno trattati in dettaglio nel Modulo 5.

Sembra evidente che non sia così banale eseguire calcoli con i tipi primitivi. Spesso conviene utilizzare solamente tipi `double` in caso di espressioni aritmetiche che coinvolgono numeri decimali. Purtroppo però, persino utilizzando solo variabili `double`, non saremo sicuri di ottenere risultati

precisi in caso di espressioni che coinvolgono più cifre decimali. Infatti, avendo i `double` una rappresentazione numerica comunque limitata, a volte devono essere arrotondati. Inoltre l'arrotondamento dipende dall'architettura della piattaforma su cui viene eseguito il calcolo.

Esiste un Java un modificatore che viene usato rarissimamente, si tratta di `strictfp`, e probabilmente la maggior parte dei programmatori Java non ne ha mai fatto uso. Dichiare le variabili `float` e `double` come `strictfp`, permetterebbe di troncare i bit che farebbero variare questi arrotondamenti su piattaforme diverse, garantendo così uniformità di risultati su tutte le piattaforme (purtroppo i risultati potrebbero comunque essere sbagliati!). Il modificatore `strictfp` può marcare non solo le variabili `float` e `double`, ma anche classi o metodi. In questi casi tutte le operazioni fatte all'interno di queste classi o metodi seguiranno le regole del troncamento di `strictfp`.

Per ottenere risultati precisi nelle operazioni è necessario utilizzare una classe della libreria Java: `BigDecimal` (package `java.math`; cfr. documentazione ufficiale) in luogo del tipo `double`.

3.2.3 Underscore in tipi di dati numerici

Per migliorare la leggibilità dei valori assegnati alle nostre variabili, è possibile usare anche i simboli di “_” (simbolo di sottolineatura o underscore). Per esempio:

```
int i = 1000000000;
int n = 0b10100001010001011010000101000101
```

Possono essere riportati come segue:

```
int i = 1_000_000_000;
int n = 0b10100001_01000101_10100001_01000101
```

Quando si vogliono utilizzare gli underscore all'interno dei numeri bisogna sapere che non è possibile usarli:

1. all'inizio o alla fine di un numero.
2. Adiacenti ad un punto decimale per i tipi di dati a virgola mobile.
3. Prima dei suffissi “F” o “L” (sia maiuscole che minuscole) che vengono usati per i cast a `float` e `long`.
4. Nelle posizioni dove ci si aspetta una stringa di caratteri.

Con qualche esempio preso direttamente dalla documentazione del JDK faremo un po' di chiarezza:

```
float piGreco = 3.14_15F;
long bytesEsadecimali = 0xFF_EC_DE_5E;
long maxLong = 0x7fff_ffff_ffff_ffffL;
long bytes = 0b11010010_01101001_10010100_10010010;
```

```
float pi1 = 3._1415F;           // Errore: violata regola 2
float pi2 = 3._1415F;           // Errore: violata regola 2
long socialSecurityNumber1 = 999_99_9999_L; // Errore: violata regola 3
int x1 = _52;                  // Errore: violata regola 1
int x2 = 5_2;
int x3 = 52_;                 // Errore: violata regola 1
int x4 = 5 ____2;
int x5 = 0_x52;                // Errore: violata regola 4
int x6 = 0x_52;                // Errore: violata regola 1
int x7 = 0x5_2;                // OK (hexadecimal literal)
int x8 = 0x52_;                // Errore: violata regola 1
int x9 = 0_52;                 // OK! Non viola la regola 1 (notare che si
                                // tratta di rappresentazione ottale)
```

3.2.4 Tipo di dato logico - booleano

Il tipo di dato `boolean` utilizza solo un bit per memorizzare un valore e gli unici valori che può immagazzinare sono `true` e `false`. Per esempio:

```
boolean b = true;
```

Vengono definiti **Literals** i valori (contenenti lettere o simboli) che vengono specificati nel codice sorgente invece che al runtime. Possono rappresentare variabili primitive o stringhe e possono apparire solo sul lato destro di un'assegnazione o come argomenti quando si invocano metodi. Non è possibile assegnare valori ai literals. Chiaramente i literals devono contenere caratteri o simboli (come per esempio il punto). Esempi di literals sono `true` e `false` (gli unici due valori assegnabili ad un tipo booleano), ma anche i valori assegnati alle stringhe, ai caratteri, e ai numeri rappresentati con simboli e lettere. Segue qualche esempio di literals:

```
boolean isBig = true;
char c = 'w';
String a = "\n";
int i = 0x1c;
float f = 3.14f;
double d = 3.14;
int unMiliardo=1_000_000_000;
```

La definizione di Literal potrebbe risultare fine a se stessa, ma è una di quelle definizioni che spesso viene menzionata nella documentazione standard.

3.2.5 Tipo di dato primitivo letterale

Il tipo `char` permette di immagazzinare caratteri (uno per volta). Utilizza l'insieme Unicode per la decodifica dei caratteri. Unicode (versione 6.2) ha tre forme:

1. UTF-8: a 8-bit che coincide con la codifica ASCII.
2. UTF-16: a 16 bit, che contiene tutti i caratteri della maggior parte degli alfabeti del mondo. Anche il `char` è 16 bit e quindi con esso siamo in grado di rappresentare praticamente tutti i caratteri più significativi esistenti.
3. UTF-32: a 32 bit che contiene codifiche di altri caratteri compresi che potrebbero essere considerati meno usati. Java supporta l'utilizzo di UTF-32 con uno stratagemma basato su un `int` (ovvero bisogna concatenare due caratteri Unicode), sarà molto raro avere bisogno di utilizzare questa forma.

Per maggiori informazioni: <http://www.unicode.org>.

Possiamo assegnare ad un `char` un qualsiasi carattere che si trova sulla nostra tastiera (ma anche il prompt Dos deve supportare il carattere richiesto). Oppure possiamo assegnare ad un `char` direttamente un valore Unicode in esadecimale che identifica univocamente un determinato carattere, anteponendo ad essa il prefisso `\u`. In qualsiasi caso, dobbiamo comprendere tra apici singoli il valore da assegnare. Ecco qualche esempio:

```
char primoCarattere = 'a';
char car ='@';
char letteraGrecaPhi ='\\u03A6'; //(lettera "Φ")
char carattereUnicodeNonIdentificato ='\\uABC8';
```

Esiste anche la possibilità di immagazzinare caratteri di escape come:

- `\n` che equivale ad andare a capo (tasto new line)
- `\\\` che equivale ad un solo `\` (tasto backslash)
- `\t` che equivale ad una tabulazione (tasto TAB)
- `\'` che equivale ad un apice singolo
- `\\"` che equivale ad un doppio apice (virgolette)

Si noti che è possibile utilizzare caratteri in espressioni aritmetiche. Per esempio possiamo sommare un `char` con un `int`.

Infatti ad ogni carattere corrisponde un numero intero. Per esempio:

```
int i = 1;
char a = 'A';
char b = (char)(a+i); // b ha valore 'B'!
```

3.3 Tipi di dati non primitivi: reference

Abbiamo già visto come istanziare oggetti da una certa classe. Dobbiamo prima dichiarare un oggetto di tale classe con una sintassi di questo tipo:

```
NomeClasse nomeOggetto;
```

per poi istanziarlo utilizzando la parola chiave `new`.

Dichiarare un oggetto quindi è del tutto simile a dichiarare un tipo di dato primitivo. Il nome che diamo ad un oggetto è detto *reference*. Infatti non si sta parlando di una variabile tradizionale, bensì di una variabile che alcuni definiscono *puntatore*. Possiamo definire un puntatore come una variabile che contiene un indirizzo in memoria. C'è una sottile e potente differenza tra la dichiarazione di un tipo di dato primitivo ed uno non primitivo. Consideriamo un esempio, partendo dalla definizione di una classe che astrae in maniera banale il concetto di data:

```
public class Data {  
    public int giorno;  
    public int mese;  
    public int anno;  
}
```

Per il nostro esempio `Data` sarà quindi un tipo di dato non primitivo (astratto). Come tipo di dato primitivo consideriamo un `double`. Consideriamo le seguenti righe di codice, supponendo che si trovino all'interno di un metodo `main()` di una classe qualsiasi:

```
double unNumero = 5.0;  
Data unGiorno = new Data();
```

Potrebbe aiutarci una schematizzazione di come saranno rappresentati i dati in memoria quando l'applicazione è in esecuzione.

In realtà riportare fedelmente queste informazioni è un'impresa ardua e se vogliamo anche inutile, in quanto programmando in Java non avremo mai a che fare direttamente con la gestione della memoria.

Ad ogni modo può essere utile immaginare la situazione in memoria con il tipo di schematizzazione, frutto di una convenzione, illustrato in Figura 3.1.

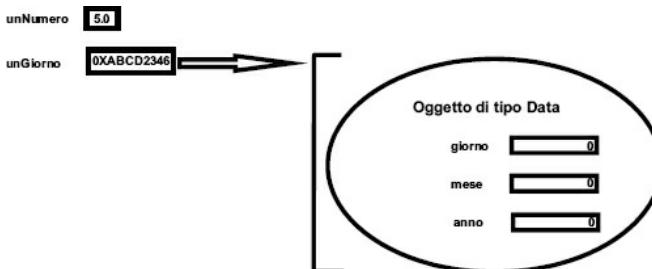


Figura 3.1 – Convenzione di schematizzazione della memoria.

La differenza pratica tra un reference ed una variabile diviene evidente nelle assegnazioni. Consideriamo il seguente frammento di codice:

```
double unNumero = 5.0;  
double unAltroNumero = unNumero;  
Data unGiorno = new Data();  
Data unAltroGiorno = unGiorno;
```

Sia per il tipo di dato primitivo sia per quello complesso, abbiamo creato un comportamento equivalente: dichiarazione ed assegnazione di un valore e riassegnazione di un altro valore.

La variabile `unAltroNumero` assumerà lo stesso valore della variabile `unNumero`, ma le due variabili rimarranno indipendenti l'una dall'altra. Il valore della variabile `unNumero` verrà infatti copiato nella variabile `unAltroNumero`. Se il valore di una delle due variabili sarà modificato in seguito, il valore dell'altra variabile non cambierà.

Il `reference` `unAltroGiorno` invece, assumerà semplicemente il valore (cioè l'indirizzo) del `reference` `unGiorno`. Ciò significa che `unAltroGiorno` punterà allo stesso oggetto cui punta `unGiorno`. La Figura 3.2 mostra la situazione rappresentata graficamente.

Quindi, se in seguito sarà apportata una qualche modifica all'oggetto comune tramite uno dei due `reference`, questa sarà verificabile anche tramite l'altro `reference`. Per intenderci:

```
unGiorno.anno
```

è sicuramente equivalente a:

```
unAltroGiorno.anno
```

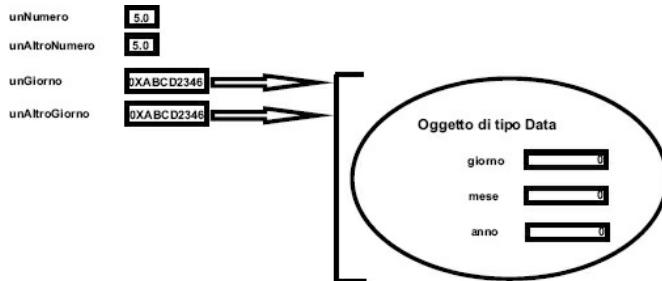


Figura 3.2 – Convenzione di schematizzazione della memoria.

3.3.1 Passaggio di parametri per valore

“Il passaggio di parametri in Java avviene **sempre** per valore”.

Quest'affermazione viene contraddetta in alcuni testi, ma basterà leggere l'intero paragrafo per fugare ogni dubbio.

Quando si invoca un metodo che come parametro prende in input una variabile, al metodo stesso

viene passato solo il valore (una copia) della variabile, che quindi rimane immutata anche dopo l'esecuzione del metodo. Per esempio consideriamo la classe:

```
public class CiProvo {  
    public void cambiaValore(int valore) {  
        valore = 1000;  
    }  
}
```

il seguente frammento di codice:

```
CiProvo ogg = new CiProvo();  
int numero = 10;  
ogg.cambiaValore(numero);  
System.out.println("il valore del numero è " + numero);
```

produrrà il seguente output:

```
il valore del numero è 10
```

Infatti il parametro `valore` del metodo `cambiaValore()`, nel momento in cui è stato eseguito il metodo, non coincideva con la variabile `numero`, bensì immagazzinava solo la copia del suo valore (10). Quindi la variabile `numero` non è stata modificata.

Stesso discorso vale per i tipi reference: viene sempre passato il valore del reference, ovvero l'indirizzo in memoria. Consideriamo la seguente classe:

```
public class CiProvoConIReference {  
    public void cambiaReference(Data data) {  
        data = new Data();  
        // Un oggetto appena istanziato  
        // ha le variabili inizializzate a valori nulli  
    }  
}
```

il seguente frammento di codice:

```
CiProvoConIReference ogg = new CiProvoConIReference();  
Data dataDiNascita = new Data();  
dataDiNascita.giorno = 14;  
dataDiNascita.mese = 4;  
dataDiNascita.anno = 2004;  
ogg.cambiaReference(dataDiNascita);  
System.out.println("Data di nascita = "  
    + dataDiNascita.giorno + "-" + dataDiNascita.mese  
    + "-" + dataDiNascita.anno );
```

produrrà il seguente output:

Valgono quindi le stesse regole anche per i reference.

Se il metodo `cambiaReference()` avesse cambiato i valori delle variabili d'istanza dell'oggetto avremmo avuto un output differente. Riscriviamo il metodo in questione:

```
public void cambiaReference(Data data){
    data.giorno = 12; // data punta allo stesso indirizzo
    data.mese = 11; // della variabile dataDiNascita
    data.anno = 2006;
}
```

Il fatto che il passaggio avvenga sempre per valore garantisce che un oggetto possa essere modificato all'interno del metodo chiamato e contemporaneamente si è certi che dopo la chiamata del metodo il reference punti sempre allo stesso oggetto.

In altri linguaggi, come il C, è permesso anche il passaggio di parametri *per riferimento*. In quel caso al metodo viene passato l'intero riferimento, non solo il suo indirizzo, con la conseguente possibilità di poterne cambiare l'indirizzamento. Questa caratteristica non è stata importata in Java, perché considerata (a ragione) una minaccia alla sicurezza. Molti virus, worm etc. sfruttano infatti la tecnica del passaggio per riferimento.

Per documentarsi sul passaggio per riferimento in C (e in generale dei puntatori) consigliamo la semplice quanto efficace spiegazione data da Bruce Eckel nel modulo 3 del suo "Thinking in C++ - 2nd Edition", gratuitamente scaricabile da <http://www.mindview.net>.

In questo caso Java ha scelto la strada della robustezza e della semplicità, favorendola rispetto alla mera potenza del linguaggio.

Alcuni autori di altri testi affermano che il passaggio di parametri in Java avviene per valore per i tipi di dato primitivi, e per riferimento per i tipi di dato complesso. Chi vi scrive ribadisce che si tratta solo di un problema di terminologia. Probabilmente, se ignorassimo il linguaggio C, anche noi daremmo un significato diverso al passaggio per riferimento. L'importante è capire il concetto senza fare confusione.

Confusione aggravata anche dalla teoria degli Enterprise JavaBeans (EJB), dove effettivamente si parla di passaggio per valore e per riferimento. Ma in quel caso ci si trova in ambiente distribuito, ed il significato cambia ancora.

3.3.2 Inizializzazione delle variabili d'istanza

Abbiamo già asserito che nel momento in cui è istanziato un oggetto tutte le sue variabili d'istanza

(che con esso condividono il ciclo di vita) vengono inizializzate ai rispettivi valori nulli. Di seguito è presentata una tabella che associa ad ogni tipo di dato il valore con cui viene inizializzata una variabile di istanza al momento della sua creazione:

Variabile	Valore
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000' (NULL)
boolean	false
Ogni tipo reference	null

3.4 Introduzione alla libreria standard

Come già accennato più volte Java possiede un'enorme e lussuosa libreria di classi standard, che costituisce uno dei punti di forza del linguaggio. Essa è organizzata in vari package (letteralmente pacchetti, fisicamente cartelle) che raccolgono le classi secondo un'organizzazione basata sul campo d'utilizzo. I principali package (meglio dire i package storici) sono:

- `java.io` contiene classi per realizzare l'input - output in Java
- `java.awt` contiene classi per realizzare interfacce grafiche, come `Button`
- `java.net` contiene classi per realizzare connessioni, come `Socket`
- `java.applet` contiene un'unica classe: `Applet`. Questa permette di realizzare applet
- `java.util` raccoglie classi d'utilità, come `Date`
- `java.lang` è il package che contiene le classi nucleo del linguaggio, come `System` e `String`

Affronteremo questi package nei moduli successivi e in alcune appendici.

Abbiamo parlato di package principali perché Sun li ha dichiarati tali per anni. Probabilmente oggi bisognerebbe aggiungere altri package a questa lista ed eliminarne alcuni.

3.4.1 Il comando `import`

Per utilizzare una classe della libreria all'interno di una classe che abbiamo intenzione di scrivere, bisogna prima importarla. Supponiamo di voler utilizzare la classe `Date` del package `java.util`. Prima di dichiarare la classe in cui abbiamo intenzione di utilizzare `Date` dobbiamo scrivere:

```
import java.util.Date;
```

oppure, per importare tutte le classi del package `java.util`:

```
import java.util.*;
```

Di default in ogni file Java è importato automaticamente tutto il package `java.lang`, senza il quale non potremmo utilizzare classi fondamentali quali `System` e `String`. Notiamo che questa è una delle caratteristiche che rende Java definibile come “semplice”. Quindi, nel momento in cui compiliamo una classe Java, il compilatore anteporrà il comando:

```
import java.lang.*;
```

alla dichiarazione della nostra classe.

L'asterisco non implica l'importazione delle classi appartenenti ai sottopackage; per esempio `import java.*` non include `java.awt.*` né `java.awt.event.*`. Quindi l'istruzione `import java.*` non importa tutti i package fondamentali.

Per dare un'idea della potenza e della semplicità di Java, viene presentata di seguito una semplicissima classe. Istanziando qualche oggetto da alcune classi del package `java.awt` (una libreria grafica a cui è dedicata l'appendice Q) ed assemblandoli con un certo criterio otterremo, con poche righe, una finestra con un pulsante. La finestra, sfruttando la libreria `java.awt`, erediterà lo stile grafico del sistema operativo su cui è eseguita. Quindi sarà visualizzato lo stile di Windows su Windows, lo stile Motif su sistema operativo Solaris e così via. Il lettore può farsi un'idea di come è possibile utilizzare la libreria standard e della sua potenza, analizzando il seguente codice:

```
import java.awt.*;

public class FinestraConBottone {
    public static void main(String args[]) {
        Frame finestra = new Frame("Titolo");
        Button bottone = new Button("Cliccami");
        finestra.add(bottone);
        finestra.setSize(200,100);
        finestra.setVisible(true);
    }
}
```

Basta conoscere un po' d'inglese per interpretare il significato di queste righe.

La classe `FinestraConBottone` è stata riportata a puro scopo didattico. La pressione del pulsante non provocherà nessuna azione, come nemmeno il tentativo di chiudere la finestra. Solo il ridimensionamento della finestra è permesso perché rientra nelle caratteristiche della classe `Frame`. Quindi, per chiudere l'applicazione bisogna spostarsi sul prompt Dos da dove la si è eseguita e terminare il processo in esecuzione mediante il comando `CTRL-C` (premere contemporaneamente i tasti “ctrl” e “c”). Se utilizzate

EJE premere il pulsante “interrompi processo”. Un’altra soluzione è bloccare il processo Java direttamente dal task manager di Windows. Anche se l’argomento vi ha incuriosito, il consiglio è di non perdere tempo nel creare interfacce grafiche inutili ed inutilizzabili: bisogna prima imparare Java! Alle interfacce grafiche e alla libreria AWT è comunque dedicata l’intera appendice Q che trovate on line.

3.4.2 La classe String

In Java le stringhe, a differenza della maggior parte dei linguaggi di programmazione, non sono array di caratteri (`char`), bensì oggetti. Le stringhe, in quanto oggetti, dovrebbero essere istanziate con la solita sintassi tramite la parola chiave `new`. Per esempio:

```
String nome = new String("Mario Rossi");
```

Abbiamo anche sfruttato il concetto di costruttore introdotto nel precedente modulo.

Java però, come spesso accade, semplifica la vita del programmatore permettendogli di utilizzare le stringhe come se si trattasse di un tipo di dato primitivo. Per esempio, possiamo istanziare una stringa nel seguente modo:

```
String nome = "Mario Rossi";
```

che è equivalente a scrivere:

```
String nome = new String("Mario Rossi");
```

Per assegnare un valore ad una stringa bisogna che esso sia compreso tra virgolette, a differenza dei caratteri per cui vengono utilizzati gli apici singoli.

Anche in questo caso possiamo apprezzare le semplificazioni che ci offre Java. Il fatto che sia permesso utilizzare una classe così importante come `String`, come se fosse un tipo di dato primitivo, ci ha permesso d’approcciare ai primi esempi di codice senza l’ulteriore “trauma” di capire come funziona un costruttore.

Il fatto che `String` sia una classe ci garantisce una serie di metodi di utilità, semplici da utilizzare e sempre disponibili, per compiere operazioni con le stringhe. Qualche esempio è rappresentato dai metodi `toUpperCase()`, che restituisce la stringa su cui è chiamato il metodo con ogni carattere maiuscolo (esiste anche il metodo `toLowerCase()`), `trim()`, che restituisce la stringa su cui è chiamato il metodo ma senza gli spazi che precedono la prima lettera e quelli che seguono l’ultima, e `equals(String)` che permette di confrontare due stringhe.

La classe `String` è chiaramente una classe particolare. Un’altra caratteristica da sottolineare è che un oggetto `String` è immutabile. I metodi di cui sopra, infatti, non vanno a modificare l’oggetto su cui sono chiamati ma, semmai, ne restituiscono un

altro. Per esempio le seguenti righe di codice:

```
String a = "giorgio";
String b = a.toUpperCase();
System.out.println(a); // a rimane immutato
System.out.println(b); // b è la stringa maiuscola
```

produrrebbero il seguente output:

```
giorgio
GIORGIO
```

3.4.3 La documentazione della libreria standard di Java

Per conoscere la classe `String` e tutte le altre classi, basta andare a consultare la documentazione: aprire il file `index.html` che si trova nella cartella API della cartella Docs del JDK. Se non trovate questa cartella, effettuate una ricerca sul disco rigido. Potreste infatti averla installata in un'altra directory. Se la ricerca fallisce, probabilmente non avete ancora scaricato la documentazione e bisogna scaricarla (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>), altrimenti potete iniziare a studiare anche un altro linguaggio di programmazione! È assolutamente fondamentale infatti che il lettore inizi da subito la sua esplorazione e conoscenza della documentazione. Il vero programmatore Java ha grande familiarità con essa e sa sfruttarne la facilità di consultazione nel modo migliore. In questo testo, a differenza di altri, saranno affrontati argomenti concernenti le classi della libreria standard, ma in maniera essenziale. Quindi non perderemo tempo nel descrivere tutti i metodi di una classe (tranne che in alcuni casi). Piuttosto ci impegheremo a capire quali sono le filosofie alla base dell'utilizzo dei vari package. Questo essenzialmente perché:

1. Riteniamo la documentazione ufficiale insostituibile.
2. Il sito Oracle ed Internet sono fonti inesauribili di informazioni ed esempi.
3. Le librerie sono in continua evoluzione.

Se utilizzate EJE è possibile consultare la documentazione direttamente da EJE. Questo a patto di installare la cartella “docs” all’interno della cartella del jdk (parallelamente a “bin”, “lib”, “jre” etc.). Altrimenti è possibile scegliere la posizione della documentazione in un secondo momento.

Molti tool di sviluppo (compreso EJE) permettono la lettura dei metodi di una classe, proponendo una lista popup ogniqualvolta lo sviluppatore utilizza l’operatore dot. È possibile quindi scrivere il metodo da utilizzare in maniera rapida, semplicemente selezionandolo da questa lista. Questo è uno dei bonus che offrono i tool, a cui è difficile rinunciare. Bisogna però sempre ricordarsi di utilizzare metodi solo dopo averne letto la documentazione. Andare a tentativi magari fidandosi del nome del metodo è una pratica assolutamente sconsigliabile. La pigrizia potrebbe costare ore di debug.

La documentazione delle Application Programming Interface (API) di Java è in formato HTML e permette una rapida consultazione. È completa e spesso esaustiva. Non è raro trovare rimandi a link online, libri o tutorial interni alla documentazione stessa. In Figura 3.3 viene riportato uno snapshot riguardante la classe `java.awt.Button`.

The screenshot shows the Java Platform Standard Edition 8 API documentation. The left sidebar lists packages like java.applet, java.awt, and java.awt.color. The main content area has tabs for OVERVIEW, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. The CLASS tab is selected. Below it are links for PREV CLASS, NEXT CLASS, FRAMES, and NO FRAMES. The SUMMARY, NESTED, FIELD, CONSTR, and METHOD sections are also visible. The central part of the page is titled "Class Button" and shows the inheritance chain: java.lang.Object > java.awt.Component > java.awt.Button. It lists implemented interfaces: ImageObserver, MenuContainer, Serializable, and Accessible. The code definition for the public class Button is shown. A descriptive text explains that this class creates a labeled button and can cause actions when pushed. Three small images below show a "Quit" button in three states: normal, active (outline darkened), and pressed (outline darkened with a mouse cursor over it). A note at the bottom states that clicking on a button generates an ActionEvent.

Figura 3.3 – Documentazione ufficiale della classe `java.awt.Button`.

Di default, la documentazione appare divisa in tre frame: in alto a sinistra vengono riportati i nomi di tutti i package, in basso a sinistra i nomi di tutte le classi e, nel frame centrale, la descrizione di ciò che è stato richiesto nel frame in basso a sinistra. Questo ipertesto rappresenta uno strumento **insostituibile** ed è invidiato dai programmati di altri linguaggi.

3.4.4 Lo strumento javadoc

Abbiamo prima accennato alla possibilità di generare documentazione in formato HTML delle nostre classi sul modello della documentazione delle classi standard di Java. Infatti esiste un trucco per la generazione automatica: lo strumento `javadoc`. Esso permette di generare ipertesti come quello della libreria standard con le informazioni sulle classi che scriveremo. Dobbiamo solamente:

1. scrivere all'interno del commento il codice accompagnato dai commenti da formattare, utilizzando commenti del terzo tipo, quelli compresi tra `/**` e `*/`;
2. utilizzare il tool `javadoc`. È molto semplice. Dal prompt, digitare:

```
javadoc nomeFile.java
```

e saranno generati automaticamente tutti i file HTML che servono, provare per credere.

Potremo generare documentazione solo per classi dichiarate `public`. Possiamo commentare classi, metodi, costruttori, variabili, costanti ed interfacce (se dichiarate `public`). Inoltre il commento deve precedere quello che si vuol commentare. Per esempio:

```
/**  
 * Questo è un metodo!  
 */  
public void metodo(){  
    . . .  
}
```

Gli IDE di sviluppo più importanti come Eclipse e Netbeans offrono una serie di utility per generare in maniera standard i commenti. In questo libro non ci dilungheremo sugli standard da usare.

Se si utilizza EJE è possibile generare la documentazione javadoc semplicemente facendo clic sul pulsante apposito. La documentazione verrà generata in una cartella "docs" creata al volo nella cartella dove si trova il file sorgente.

3.5 Gli array in Java

Un array è una collezione indicizzata di dati primitivi, o di reference, o di altri array. Gli array permettono di utilizzare un solo nome per individuare una collezione costituita da vari elementi, che saranno accessibili tramite indici interi. In Java gli array sono, in quanto collezioni, oggetti.

Per utilizzare un array bisogna passare attraverso tre fasi: dichiarazione, creazione ed inizializzazione (come per gli oggetti).

3.5.1 Dichiarazione

Di seguito presentiamo due dichiarazioni di array. Nella prima dichiariamo un array di `char` (tipo primitivo), nella seconda dichiariamo un array di istanze di `Button` (classe appartenente al package `java.awt`):

```
char alfabeto [];  
        oppure  
        char [] alfabeto;  
  
Button bottoni [];  
        oppure  
        Button [] bottoni;
```

In definitiva per dichiarare un array è necessario posporre (oppure anteporre) una coppia di parentesi quadre all'identificatore.

3.5.2 Creazione

Un array è un oggetto speciale in Java e, in quanto tale, va istanziato in modo speciale. La sintassi è la seguente:

```
alfabeto = new char[21];
bottoni = new Button[3];
```

Come si può notare, è obbligatorio specificare al momento dell'istanza dell'array la dimensione dell'array stesso. A questo punto però tutti gli elementi dei due array sono inizializzati automaticamente ai relativi valori nulli. Vediamo allora come inizializzare esplicitamente gli elementi dell'array.

3.5.3 Inizializzazione

Per inizializzare un array, bisogna inizializzarne ogni elemento singolarmente:

```
alfabeto [0] = 'a';
alfabeto [1] = 'b';
alfabeto [2] = 'c';
alfabeto [3] = 'd';
. . .
alfabeto [20] = 'z';

bottoni [0] = new Button();
bottoni [1] = new Button();
bottoni [2] = new Button();
```

L'indice di un array inizia sempre da zero. Quindi un array dichiarato di 21 posti, avrà come indice minimo 0 e massimo 20 (un array di dimensione n implica il massimo indice a $n-1$).

Il lettore avrà sicuramente notato che può risultare alquanto scomodo inizializzare un array in questo modo, per di più dopo averlo prima dichiarato ed istanziato. Ma Java ci viene incontro dando la possibilità di eseguire tutti e tre i passi principali per creare un array tramite una particolare sintassi che di seguito presentiamo:

```
char alfabeto [] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'z'};
Button buttoni [] = {new Button(), new Button(), new Button()} ;
```

Si noti la differenza tra un array dichiarato di tipo di dato primitivo o complesso. Il primo contiene direttamente i suoi elementi. Il secondo contiene solo reference, non gli elementi stessi.

Gli array definiscono anche una variabile chiamata `length` che restituisce la dimensione effettiva dell'array stesso. Quindi:

```
alfabeto.length
```

varrà 21.

Solitamente uno dei vantaggi che porta l'uso di array è sfruttare l'indice all'interno di cicli, che

però verranno trattati nel prossimo modulo.

In generale gli array sono utilizzati relativamente poco in Java. Ad essi sono preferiti gli oggetti della libreria Collections che saranno trattati approfonditamente nei moduli successivi.

3.5.4 Array multidimensionali

Esistono anche array multidimensionali, che sono array di array. A differenza della maggior parte degli altri linguaggi di programmazione, in Java quindi, un array bidimensionale non deve per forza essere rettangolare. Di seguito è presentato un esempio:

```
int arrayNonRettangolare [][] = new int[4] [];
arrayNonRettangolare [0] = new int[2];
arrayNonRettangolare [1] = new int[4];
arrayNonRettangolare [2] = new int[6];
arrayNonRettangolare [3] = new int[8];
arrayNonRettangolare [0][0] = 1;
arrayNonRettangolare [0][1] = 2;
arrayNonRettangolare [1][0] = 1;
. . . . .
arrayNonRettangolare [3][7] = 10;
```

oppure, equivalentemente:

```
int arrayNonRettangolare [][] = {
{1,2},
{1,0,0,0},
{0,0,0,0,0,0},
{0,0,0,0,0,0,0,10}
};
```

Si noti che nella prima riga quando abbiamo istanziato l'oggetto `arrayNonRettangolare`, abbiamo specificato solo la prima dimensione (4). Questo significa che ci saranno quattro elementi di questo array, ed ogni elemento sarà a sua volta un altro array di lunghezza ancora da scegliere. Attenzione che una rappresentazione grafica di un array è solo una convenzione che ci aiuta a schematizzare il concetto. Non si dovrebbe quindi parlare di righe e colonne per un array bidimensionale, le righe e le colonne sono solo nella nostra mente, in realtà Java è capace anche di creare array a dieci dimensioni, che noi non potremmo mai rappresentare graficamente.

3.5.5 String args[]

Di solito, quando si inizia ad imparare un nuovo linguaggio di programmazione, uno dei primi argomenti che l'aspirante sviluppatore impara a gestire, è l'input/output nelle applicazioni. Quando

invece s'approccia a Java, rimane misterioso per un certo periodo il “comando” di output:

```
System.out.println("Stringa da stampare");
```

e resta sconosciuta per un lungo periodo anche un’istruzione che permetta di acquisire dati in input! Ciò è dovuto ad una ragione ben precisa: le classi che permettono di realizzare operazioni di input/output fanno parte del package `java.io` della libreria standard, a cui è dedicato più avanti un apposito modulo. Questo package è stato progettato con una filosofia ben precisa, basata sul pattern *Decorator* (per informazioni sui pattern cfr. appendice D). Il risultato è un’iniziale difficoltà d’approccio all’argomento, compensata però da una semplicità ed efficacia finale. Per esempio, a un aspirante programmatore può risultare difficoltoso comprendere le ragioni per cui, per stampare una stringa a video, i creatori di Java hanno implementato un meccanismo tanto complesso (`System.out.println()`). Per un programmatore Java invece è molto semplice utilizzare gli stessi metodi per eseguire operazioni di output complesse, come scrivere in un file o mandare messaggi in rete via socket. Rimandiamo il lettore al modulo relativo all’input/output per i dettagli. Per non anticipare troppo i tempi, presentiamo intanto un procedimento che permette di dotare di un minimo d’interattività le nostre prime applicazioni. Quando codifichiamo il metodo `main()`, il programmatore è obbligato a fornire una firma che definisce come parametro in input un array di stringhe (di solito chiamato `args`). Questo array immagazzinerà stringhe da riga di comando nel modo seguente. Se per eseguire la nostra applicazione, invece di scrivere a riga di comando:

```
java NomeClassDelMain
```

scrivessimo:

```
java NomeClassDelMain Andrea De Sio Cesari
```

all’interno della nostra applicazione avremmo a disposizione la stringa `args[0]` che ha come valore `Andrea`, la stringa `args[1]` che ha come valore `De`, la stringa `args[2]` che ha come valore `Sio`, e la stringa `args[3]` che ha come valore `Cesari`. Potremmo anche scrivere:

```
java NomeClassDelMain Andrea "De Sio Cesari"
```

in questo modo, all’interno dell’applicazione potremmo utilizzare solamente la stringa `args[0]` che ha come valore `Andrea`, e la stringa `args[1]` che ha come valore `De Sio Cesari`.

Se come editor state utilizzando EJE, allora per poter passare parametri da riga di comando bisogna eseguire l’applicazione dal menu “sviluppo – esegui con argomenti” (in inglese “build - execute with args”). In alternativa è possibile usare la scorciatoia costituita dalla pressione dei tasti SHIFT-F9. Verrà presentata una maschera per inserire gli argomenti (e solo gli argomenti).

In pratica a seconda di quanti argomenti passiamo da riga di comando, la JVM istanzierà l'array di stringhe `args` di lunghezza pari al numero di argomenti passati, e lo inizializzerà come abbiamo detto. Per esempio il seguente codice:

```
public class TestArgs {  
    public static void main(String args[]) {  
        System.out.println(args[0]);  
    }  
}
```

stamperà solo il primo eventuale argomento passato da riga di comando. Se però non passeremo un argomento da riga di comando, il programma terminerà con la seguente eccezione:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0 at  
TestArgs.main(TestArgs.java:3)
```

Studieremo le eccezioni approfonditamente nel modulo 8. Ma già dal prossimo modulo dove studieremo cicli e condizioni, impareremo a scrivere programmi che non si interrompono così facilmente.

Riepilogo

In questo modulo abbiamo introdotto alcune caratteristiche fondamentali del linguaggio e imparato ad utilizzare alcune convenzioni standard per il codice. Abbiamo visto l'importanza dell'**indentazione del codice** e delle **convenzioni per gli identificatori**. Abbiamo non solo studiato gli otto **tipi di dati primitivi** di Java, ma anche alcuni dei problemi relativi ad essi, e i concetti di **casting** e **promotion**. Fondamentale è stata la discussione sul concetto di **reference**, che il lettore deve aver appreso correttamente per non incontrare problemi nel seguito del suo studio. Inoltre abbiamo introdotto la **libreria standard**, la sua **documentazione**, il comando `javadoc`, e abbiamo trattato una classe fondamentale come la classe `String`. Per completezza abbiamo anche descritto gli **array** con le loro caratteristiche ma ne apprezzeremo l'utilità più avanti.

Esercizi modulo 3

Esercizio 3.a)

Scrivere un semplice programma che svolga le seguenti operazioni aritmetiche correttamente, scegliendo accuratamente i tipi di dati da utilizzare per immagazzinare i risultati di esse.

Una divisione (usare il simbolo `/`) tra due interi `a = 5`, e `b = 3`. Immagazzinare il risultato in una variabile `r1`, scegliendo il tipo di dato adeguato.

Una moltiplicazione (usare il simbolo `*`) tra un `char c = 'a'`, ed uno `short s = 5000`. Immagazzinare il risultato in una variabile `r2`, scegliendo il tipo di dato adeguato.

Una somma (usare il simbolo `+`) tra un `int i = 6` ed un `float f = 3.14F`. Immagazzinare il risultato in una variabile `r3`, scegliendo il tipo di dato adeguato.

Una sottrazione (usare il simbolo `-`) tra `r1`, `r2` e `r3`. Immagazzinare il risultato in una variabile `r4`, scegliendo il tipo di dato adeguato.

Verificare la correttezza delle operazioni stampandone i risultati parziali ed il risultato finale. Tenere presente la promozione automatica nelle espressioni e utilizzare il casting propriamente.

Basta una classe con un `main()` che svolga le operazioni.

Esercizio 3.b)

Scrivere un programma con i seguenti requisiti.

Utilizza una classe `Persona` che dichiara le variabili `nome`, `cognome`, `eta` (età). Si dichiari inoltre un metodo `dettagli()` che restituisca in una stringa le informazioni sulla persona in questione. Ricordarsi di utilizzare le convenzioni e le regole descritte in questo modulo.

Utilizza una classe `Principale` che, nel metodo `main()`, istanzia due oggetti chiamati `personal1` e `personal2` della classe `Persona`, inizializzando per ognuno di essi i relativi campi adoperando l'operatore dot.

Dichiarare un terzo reference (`personal3`) che punti ad uno degli oggetti già istanziati. Controllare che effettivamente `personal3` punti all'oggetto voluto, stampando i campi di `personal3` sempre mediante l'operatore dot.

Commentare adeguatamente le classi realizzate e sfruttare lo strumento `javadoc` per produrre la relativa documentazione.

Nella documentazione standard di Java sono usate tutte le regole e le convenzioni descritte in questo capitolo. Basti osservare che `String`, essendo una classe, inizia con lettera maiuscola. Si può concludere che anche `System` è una classe.

Esercizio 3.c) Array, Vero o Falso:

1. Un array è un oggetto e quindi può essere dichiarato, istanziato ed inizializzato.
2. Un array bidimensionale è un array i cui elementi sono altri array.
3. Il metodo `length` restituisce il numero degli elementi di un array.
4. Un array non è ridimensionabile.
5. Un array è eterogeneo di default.
6. Un array di interi può contenere come elementi byte, ovvero le seguenti righe di codice non producono errori in compilazione:

```
int arr [] = new int[2];
byte a = 1, b=2;
arr [0] = a;arr [1] = b;
```

7. Un array di interi può contenere come elementi `char`, ovvero le seguenti righe di codice non producono errori in compilazione:

```
char a = 'a', b = 'b';
int arr [] = {a,b};
```

8. Un array di stringhe può contenere come elementi `char`, ovvero le seguenti righe di codice non producono errori in compilazione:

```
String arr [] = {'a' , 'b'};
```

9. Un array di stringhe è un array bidimensionale, perché le stringhe non sono altro che array di caratteri. Per esempio il seguente è un array bidimensionale:

```
String arr [] = {"a" , "b"};
```

10. Se abbiamo il seguente array bidimensionale:

```
int arr [][] = {
{1, 2, 3},
{1,2},
{1,2,3,4,5}
};
```

risulterà che:

```
arr.length = 3;
arr[0].length = 3;
arr[1].length = 2;
arr[2].length = 5;
arr[0][0] = 1;
arr[0][1] = 2;
arr[0][2] = 3;
arr[1][0] = 1;
arr[1][1] = 2;
arr[1][2] = 3;
arr[2][0] = 1;
arr[2][1] = 2;
arr[2][2] = 3;
arr[2][3] = 4;
arr[2][4] = 5;
```

Soluzioni esercizi modulo 3

Esercizio 3.a)

```
public class Esercizio3A {
public static void main (String args[]) {
    int a = 5, b = 3;
double r1 = (double)a/b;
System.out.println("r1 = " + r1);
char c = 'a';
short s = 5000;
```

```
int r2 = c*s;
System.out.println("r2 = " + r2);
int i = 6;
float f = 3.14F;
float r3 = i + f;
System.out.println("r3 = " + r3);
double r4 = r1 - r2 - r3;
System.out.println("r4 = " + r4);
}
}
```

Esercizio 3.b)

```
public class Persona {
public String nome;
public String cognome;
public int eta;
public String dettagli() {
return nome + " " + cognome + " anni " + eta;
}
}

public class Principale {
public static void main (String args []) {
Persona personal = new Persona();
Persona persona2 = new Persona();
personal.nome = "Mario";
personal.cognome = "Rossi";
personal.eta = 30;
System.out.println("personal "+personal.dettagli());
persona2.nome = "Giuseppe";
persona2.cognome = "Verdi";
persona2.eta = 40;
System.out.println("persona2 "+persona2.dettagli());
Persona persona3 = personal;
System.out.println("persona3 "+persona3.dettagli());
}
}
```

Esercizio 3.c) Array, Vero o Falso:

- 1. Vero.**
- 2. Vero.**
- 3. Falso,** la variabile `length` restituisce il numero degli elementi di un array.
- 4. Vero.**
- 5. Falso.**

- 6. Vero**, un `byte` (che occupa solo 8 bit) può essere immagazzinato in una variabile `int` (che occupa 32 bit).
- 7. Vero**, un `char` (che occupa 16 bit) può essere immagazzinato in una variabile `int` (che occupa 32 bit).
- 8. Falso**, un `char` è un tipo di dato primitivo e `String` è una classe. I due tipi di dati non sono compatibili.
- 9. Falso**, in Java la stringa è una oggetto istanziato dalla classe `String` e non un array di caratteri.
- 10. Falso**, tutte le affermazioni sono giuste tranne `arr[1][2] = 3`; perché questo elemento non esiste.

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper utilizzare le convenzioni per il codice Java (unità 3.1)	<input type="checkbox"/>	
Conoscere e saper utilizzare tutti i tipi di dati primitivi (unità 3.2)	<input type="checkbox"/>	
Saper gestire casting e promotion (unità 3.2)	<input type="checkbox"/>	
Saper utilizzare i reference e capirne la filosofia (unità 3.3)	<input type="checkbox"/>	
Iniziare ad esplorare la documentazione della libreria standard di Java (unità 3.4)	<input type="checkbox"/>	
Saper utilizzare la classe String (unità 3.4)	<input type="checkbox"/>	
Saper definire, creare, e inizializzare gli array (unità 3.5)	<input type="checkbox"/>	
Saper commentare il proprio codice ed essere in grado di utilizzare lo strumento javadoc per produrre documentazione tecnica esterna (unità 3.1, 3.4)	<input type="checkbox"/>	

Note:

Operatori e gestione del flusso di esecuzione

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Conoscere e saper utilizzare i vari operatori (unità 4.1).
- ✓ Conoscere e saper utilizzare i costrutti di programmazione semplici (unità 4.2, 4.3).
- ✓ Conoscere e saper utilizzare i costrutti di programmazione avanzati (unità 4.2, 4.4).

In questo modulo finalmente affronteremo tutti gli argomenti che ci permetteranno di iniziare a programmare. Parleremo degli operatori e dei costrutti di programmazione (cicli e condizioni).

4.1 Operatori di base

Di seguito è presentata una lista completa degli operatori che Java mette a disposizione.

Java eredita in blocco tutti gli operatori del linguaggio C e quindi, per alcuni di essi, l'utilizzo è alquanto raro. Dunque non analizzeremo in dettaglio tutti gli operatori.

4.1.1 Operatore d'assegnazione

L'operatore = permette di assegnare un valore ad una variabile. È possibile assegnare valori diversi ad una variabile durante l'esecuzione del codice. Per esempio è lecito effettuare le seguenti assegnazioni:

```
int variabile1 = 1;
int variabile2 = 2;
variabile1 = variabile2;
```

dopo queste istruzioni, il valore intero 2 sarà immagazzinato sia in `variabile1` che in `variabile2`.

4.1.2 Operatori aritmetici

La seguente tabella riassume gli operatori aritmetici semplici definiti dal linguaggio:

Descrizione	Operatore
Somma	+
Sottrazione	-

Moltiplicazione	*
Divisione	/
Modulo	%

L'unico operatore che può risultare poco familiare al lettore è l'operatore modulo. Il risultato dell'operazione modulo tra due numeri coincide con il resto della divisione fra essi. Per esempio:

```
5 % 3 = 2
10 % 2 = 0
100 % 50 = 0
```

Gli operatori + e - possono essere usati anche come operatore unari (ovvero si possono applicare ad un solo operando) per specificare il segno di un numero. Per esempio le seguenti linee sono perfettamente valide:

```
int i = -1;
int j = +1;
```

Java ha ereditato dalla sintassi del linguaggio C anche altri operatori, sia binari (con due operandi) che unari (con un solo operando). Alcuni di essi, oltre a svolgere un'operazione, assegnano anche il valore del risultato ad una variabile utilizzata nell'operazione stessa:

Descrizione	Operatore
Somma ed assegnazione	+=
Sottrazione ed assegnazione	-=
Moltiplicazione ed assegnazione	*=
Divisione ed assegnazione	/=
Modulo ed assegnazione	%=

Quindi se abbiamo:

```
int i = 5;
```

scrivere:

```
i = i + 2;
```

è equivalente a scrivere:

```
i += 2;
```

4.1.3 Operatori (unari) di pre e post-incremento (e decremento)

La seguente tabella descrive questa singolare tipologia di operatori:

Descrizione	Operatore	Esempio
Pre-incremento di un'unità	++	++i

Pre-decremento di un'unità	--	--i
Post-incremento di un'unità	++	i++
Post-decremento di un'unità	--	i--

Se vogliamo incrementare di una sola unità una variabile numerica, possiamo equivalentemente scrivere:

```
i = i + 1;
```

oppure:

```
i += 1;
```

ma anche:

```
i++;
```

oppure:

```
++i;
```

ottenendo comunque lo stesso risultato. Infatti, in tutti i casi, otterremo che il valore della variabile *i* è stato incrementato di un'unità ed assegnato nuovamente alla variabile stessa. Quindi anche questi operatori svolgono due compiti (incremento ed assegnazione). Parleremo di operatore di pre-incremento nel caso in cui anteponiamo l'operatore d'incremento `++` alla variabile. Parleremo, invece, di operatore di post-incremento nel caso in cui posponiamo l'operatore di incremento alla variabile. La differenza tra questi due operatori composti consiste essenzialmente nelle priorità che essi hanno rispetto all'operatore di assegnazione. L'operatore di pre-incremento ha maggiore priorità dell'operatore di assegnazione `=`. L'operatore di post-incremento ha minor priorità rispetto all'operatore di assegnazione `=`. Le stesse regole valgono per gli operatori di decremento.

Facciamo un paio di esempi per rendere evidente la differenza tra i due operatori. Il seguente codice utilizza l'operatore di pre-incremento:

```
x = 5;
y = ++x;
```

Dopo l'esecuzione delle precedenti istruzioni avremo che:

```
x = 6
y = 6
```

Il seguente codice invece utilizza l'operatore di post-incremento:

```
x = 5;
y = x++;
```

In questo caso avremo che:

```
x = 6  
y = 5
```

4.1.4 Operatori bitwise

La seguente tabella mostra tutti gli operatori bitwise (ovvero che eseguono operazioni direttamente sui bit) definiti in Java:

Descrizione	Operatore
NOT	<code>~</code>
AND	<code>&</code>
OR	<code> </code>
XOR	<code>^</code>
Shift a sinistra	<code><<</code>
Shift a destra	<code>>></code>
Shift a destra senza segno	<code>>>></code>
AND e assegnazione	<code>&=</code>
OR e assegnazione	<code> =</code>
XOR e assegnazione	<code>^=</code>
Shift a sinistra e assegnazione	<code><<=</code>
Shift a destra e assegnazione	<code>>>=</code>
Shift a destra senza segno e assegnazione	<code>>>>=</code>

Tutti questi operatori binari sono molto efficienti giacché agiscono direttamente sui bit, ma in Java si utilizzano raramente. Infatti in Java non esiste l'aritmetica dei puntatori e, di conseguenza, lo sviluppatore non è abituato a “pensare in bit”. L'operatore NOT `~` è un operatore unario dato che si applica ad un solo operando. Per esempio, sapendo che la rappresentazione binaria di 1 è `00000001`, avremo che `~1` varrà `11111110` ovvero `-2`.

Tale operatore, applicato ad un numero intero, capovolgerà la rappresentazione dei suoi bit scambiando tutti gli 0 con 1 e viceversa.

Gli operatori `&`, `|`, `^`, si applicano a coppie di operandi, e svolgono le relative operazioni logiche di conversioni di bit riassunte nella seguente tabella della verità:

Operando1	Operando2	Op1 AND Op2	Op1 OR Op2	Op1 XOR Op2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Gli operatori di shift (operatori di scorrimento) provocano lo scorrimento di un certo numero di bit verso una determinata direzione, quella della rappresentazione binaria del dato in questione. Il numero dei bit da scorrere è rappresentato dall'operando a destra dell'operazione. I bit che dopo lo

scorrimento si trovano al di fuori della rappresentazione binaria del numero vengono eliminati. I bit che invece “rimangono vuoti” vengono riempiti con i valori 0 oppure 1 a seconda del caso. In particolare, lo scorrimento a sinistra provoca un riempimento con i valori 0 dei bit lasciati vuoti sulla destra della rappresentazione binaria del numero. Anche lo scorrimento a destra senza segno riempie i bit lasciati vuoti con degli 0. Lo scorrimento a destra con segno, invece, provoca il riempimento di 0 oppure di 1, a seconda che l’ultima cifra a sinistra prima dello scorrimento (bit del segno) sia 0 oppure 1, ovvero che la cifra prima dello scorrimento sia positiva o negativa.

```
byte a = 35; //rappresentazione binaria 00100011
```

e shiftiamo (scorriamo) a destra di due posizioni:

```
a = a >> 2;
```

avremo che:

```
a = 8 // rappresentazione binaria 00001000
```

Se invece abbiamo:

```
byte b = -8; //rappresentazione binaria 11111000
```

e shiftiamo b di una posizione:

```
b = b >> 1;
```

```
b = -4 //rappresentazione binaria 11111100
```

Facciamo ora un esempio di scorrimento a destra senza segno:

```
int a = -1;  
a = a >>> 24;  
11111111111111111111111111111111 ovvero -1  
>>> 24  
000000000000000000000000011111111 ovvero 255
```

Ricordiamo che la promozione automatica delle espressioni avviene per ogni operatore binario e quindi anche per l'operatore di scorrimento a destra senza segno.

L'operazione di scorrimento a destra equivale a dividere l'operando di sinistra per 2 elevato all'operando situato alla destra nell'espressione. Il risultato viene arrotondato per difetto nelle operazioni con resto. Ovvero: $op1 \gg op2$ equivale a $op1 / op2$

diviso (2 elevato a op2) Similmente l'operazione di scorrimento a sinistra equivale a moltiplicare l'operando di sinistra per 2 elevato all'operando situato sulla destra dell'operazione. Ovvero: op1 << op2 equivale a op1 moltiplicato (2 elevato a op2).

4.1.5 Operatori relazionali o di confronto

Il risultato delle operazioni basate su operatori relazionali è sempre un valore boolean, ovvero true o false.

Operatore	Simbolo	Applicabilità
Uguale a	==	Tutti i tipi
Diverso da	!=	Tutti i tipi
Maggiore	>	Solo i tipi numerici
Minore	<	Solo i tipi numerici
Maggiore o uguale	>=	Solo i tipi numerici
Minore o uguale	<=	Solo i tipi numerici

Un classico errore dell'aspirante programmatore è scrivere = in luogo di ==.

Se confrontiamo due reference con l'operatore == il risultato risulterà true se e solo se i due reference puntano allo stesso oggetto, altrimenti false. Infatti viene confrontato sempre il valore delle variabili in gioco. Il valore di una variabile reference, come abbiamo visto nel modulo precedente, è l'indirizzo in memoria dell'oggetto a cui punta.

La classe string, di cui abbiamo già parlato nel precedente modulo, gode di una singolare particolarità che si può notare con l'utilizzo dell'operatore ==. Abbiamo visto come sia possibile istanziare la classe string come un tipo di dato primitivo, per esempio:

```
String linguaggio = "Java";
```

Le istanze così ottenute, però, sono trattate diversamente da quelle che vengono istanziate con la sintassi tradizionale:

```
String linguaggio = new String("Java");
```

Infatti le istanze che vengono create senza la parola chiave new, come se string fosse un tipo di dato primitivo, vengono poi poste in un speciale pool di stringhe (un insieme speciale) dalla Java Virtual Machine e riutilizzate, al fine di migliorare le prestazioni. Questo significa che il seguente frammento di codice:

```
String a = "Java";
String b = "Java";
String c = new String("Java");
System.out.println(a==b);
System.out.println(b==c);
```

produrrà il seguente output:

```
true  
false
```

Infatti `a` e `b` punteranno esattamente allo stesso oggetto. Mentre nel caso di `c`, avendo utilizzato la parola chiave `new`, verrà creato un oggetto ex novo, anche se con lo stesso contenuto dei precedenti. Ricordiamo che il giusto modo per confrontare due stringhe rimane l'utilizzo del metodo `equals()`. Infatti il seguente frammento di codice:

```
System.out.println(a.equals(b));  
System.out.println(b.equals(c));
```

produrrà il seguente output:

```
true  
true
```

A tal proposito è bene ricordare che la classe `String` definisce un metodo chiamato `intern()`, che prova a recuperare proprio l'oggetto `String` dal pool di stringhe, utilizzando il confronto che fornisce il metodo `equals()`. Nel caso nel pool non esista la stringa desiderata, questa viene aggiunta, e viene restituito un reference ad essa. In pratica, date due stringhe `t` ed `s`:

```
s.intern() == t.intern()
```

è true se e solo se

```
s.equals(t)
```

vale true.

4.1.6 Operatori logico – booleani

Quelli seguenti sono operatori che utilizzano solo operandi di tipo booleano. Il risultato di un'operazione basata su tali operatori è di tipo **boolean**:

Descrizione	Operatore
NOT logico	!
AND logico	&
OR logico	
XOR logico	^
Short circuit AND	&&
Short circuit OR	
AND e assegnazione	&=
OR e assegnazione	=
XOR e assegnazione	^=

È facile trovare punti di contatto tra la precedente lista di operatori e la lista degli operatori bitweise. Gli operandi a cui si applicano gli operatori booleani però possono essere solo di tipo booleano.

È consuetudine utilizzare le versioni short circuit (“corto circuito”) di AND ed OR. Ad esempio, la seguente riga di codice mostra come avvantaggiarsi della valutazione logica di corto circuito:

```
boolean flag = ( (a != 0) && (b/a > 10) )
```

Affinché l'espressione tra parentesi sia vera, bisogna che entrambi gli operandi dell'AND siano veri. Se il primo tra loro è in partenza falso, non ha senso andare a controllare la situazione dell'altro operando. In questo caso sarebbe addirittura dannoso perché porterebbe ad una divisione per zero (errore riscontrabile solo al runtime e non in fase di compilazione). Quest'operatore short circuit, a differenza della sua versione tradizionale (`&`), fa evitare il secondo controllo in caso di fallimento del primo. Equivalentemente l'operatore short circuit `||`, nel caso la prima espressione da testare risultasse verificata, convalida l'intera espressione senza alcuna altra (superflua) verifica.

4.1.7 Concatenazione di stringhe con +

In Java l'operatore `+`, oltre che essere un operatore aritmetico, è anche un operatore per concatenare stringhe. Per esempio il seguente frammento di codice:

```
String nome = "James ";
String cognome = "Gosling";
String nomeCompleto = "Mr. " + nome + cognome;
```

farà in modo che la stringa `nomeCompleto`, abbia come valore `Mr. James Gosling`. Se concateniamo mediante il `+` un qualsiasi tipo di dato con una stringa, il tipo di dato sarà automaticamente convertito in stringa, e ciò può spesso risultare utile.

Il meccanismo di concatenazione di stringhe viene in realtà gestito dietro le quinte dal compilatore Java. Esso trasformerà istruzioni come quelle dell'esempio, in codice che sfrutta la classe `StringBuilder`. Si tratta di una classe che rappresenta una stringa ridimensionabile, mettendo a disposizione un metodo come `append()` (in italiano “aggiungi”). In pratica, il codice del precedente esempio, verrà trasformato dal compilatore nel seguente codice:

```
String nomeCompleto = new StringBuilder().
append("Mr.").
append(nome).append(cognome).toString();
```

4.1.8 Priorità degli operatori

Nella seguente tabella sono riportati, in ordine di priorità, tutti gli operatori di Java. Alcuni di essi non sono ancora stati trattati.

separatori	. () ; ,
da sx a dx	++ -- + - ~ ! (tipi di dati)
da sx a dx	* / %
da sx a dx	+ -
da sx a dx	<< >> >>>
da sx a dx	< > <= >= instanceof
da sx a dx	== !=
da sx a dx	&
da sx a dx	^
da sx a dx	
da sx a dx	&&
da sx a dx	
da dx a sx	? :
da dx a sx	= *= /= %= += -= <<= >>= >>>= &= ^= =

Fortunatamente non è necessario conoscere a memoria tutte le priorità per programmare. Nell'incertezza è sempre possibile utilizzare le parentesi tonde così come faremmo nell'aritmetica tradizionale. Non potendo usufruire di parentesi quadre e graffe (poiché in Java queste sono adoperate per altri scopi) sostituiremo il loro utilizzo sempre con parentesi tonde.

Quindi, se abbiamo le seguenti istruzioni:

```
int a = 5 + 6 * 2 - 3;
int b = (5 + 6) * (2 - 3);
int c = 5 + (6 * (2 - 3));
```

le variabili `a`, `b` e `c` varranno rispettivamente 14, -11 e -1.

4.2 Gestione del flusso di esecuzione

In ogni linguaggio di programmazione esistono costrutti che permettono all'utente di controllare la sequenza delle istruzioni immesse. Essenzialmente possiamo dividere questi costrutti in due categorie principali:

- ❑ **Condizioni (o strutture di controllo decisionali):** durante la fase di runtime permettono una scelta tra l'esecuzione di istruzioni diverse, a seconda che sia verificata una specificata condizione.
- ❑ **Cicli (strutture di controllo iterative):** in fase di runtime permettono di decidere il numero di esecuzioni di determinate istruzioni.

In Java le condizioni che si possono utilizzare sono essenzialmente due: il costrutto `if` ed il costrutto `switch`, cui si aggiunge l'**operatore ternario**. I costrutti di tipo ciclo invece sono quattro: `while`, `for`, `do` (detto anche `do-while`) e il **ciclo for migliorato** (detto anche `foreach`). Tutti i costrutti possono essere annidati. I costrutti principali (almeno da un punto di vista storico) sono la condizione `if` ed il ciclo `while`. Un programmatore in grado di utilizzare questi due costrutti sarà in grado di codificare un qualsiasi tipo di istruzione. La sintassi di questi due costrutti è alquanto banale

e per questo vengono anche detti *costrutti di programmazione semplici*.

4.3 Costrutti di programmazione semplici

In questo paragrafo studieremo i costrutti `if` e `while`. Inoltre introdurremo un operatore che non abbiamo ancora incontrato: l'operatore ternario.

4.3.1 Il costrutto `if`

Questa condizione consente di prendere semplici decisioni basate su valori immagazzinati. In fase di runtime la Java Virtual Machine testa un'espressione booleana `e`, a seconda che essa risulti vera o falsa, esegue un certo blocco di istruzioni oppure no. Un'espressione booleana è un'espressione che come risultato può restituire solo valori di tipo `boolean`, vale a dire `true` o `false`. Essa di solito si avvale di operatori di confronto e, se necessario, di operatori logici. La sintassi è la seguente:

```
if (espressione-boleana) istruzione;
```

per esempio:

```
if (numeroLati == 3)
    System.out.println("Questo è un triangolo");
```

Nell'esempio, l'istruzione di stampa sarebbe eseguita se e solo se la variabile `numeroLati` avesse valore 3. In quel caso l'espressione booleana `numeroLati == 3` varrebbe `true` e quindi sarebbe eseguita l'istruzione che segue l'espressione. Se invece l'espressione risultasse `false`, sarebbe eseguita direttamente la prima eventuale istruzione che segue l'istruzione di stampa. Possiamo anche estendere la potenzialità del costrutto `if` mediante la parola chiave `else`:

```
if (espressione-boleana) istruzione1;
else istruzione2;
```

per esempio:

```
if (numeroLati == 3)
    System.out.println("Questo è un triangolo");
else
    System.out.println("Questo non è un triangolo");
```

`if` potremmo tradurlo con “se”; `else` con “altrimenti”. In pratica, se l'espressione booleana è vera verrà stampata la stringa “Questo è un triangolo”; se è falsa verrà stampata la stringa “Questo non è un triangolo”.

Possiamo anche utilizzare blocchi di codice, con il seguente tipo di sintassi:

```
if (espressione-boleana) {
    istruzione_1;
```

```
.....;
istruzione_k;
} else {
istruzione_k+1;
.....;
istruzione_n;
}
```

ed anche comporre più costrutti nel seguente modo

```
if (espressione-boleana) {
istruzione_1;
....;
istruzione_k;
} else if (espressione-boleana){
istruzione_k+1;
....;
istruzione_j;
} else if (espressione-boleana){
istruzione_j+1;
....;
istruzione_h;
} else {
istruzione_h+1;
....;
istruzione_n;
}
```

Possiamo anche annidare questi costrutti. I due seguenti frammenti di codice possono sembrare equivalenti. In realtà il frammento di codice a sinistra mostra un **if** che annida un costrutto **if – else**. Il frammento di codice a destra invece mostra un costrutto **if – else** che annida un costrutto **if**:

```
...
if (x != 0)
    if (y < 10)
        z = 5;
    else
        z = 7;
...
```

```
...
if (x != 0) {
    if (y < 10)
        z = 5;
} else
    z = 7;
...
```

Si consiglia sempre di utilizzare un blocco di codice (mediante l'utilizzo di parentesi graffe) per circondare anche un'unica istruzione. Questa pratica oltre ad aggiungere leggibilità consente anche di aggiungere istruzioni in un secondo momento, come spesso succede.

4.3.2 L'operatore ternario

Esiste un operatore non ancora trattato qui, che qualche volta può sostituire il costrutto `if`. Si tratta del cosiddetto **operatore ternario** (detto anche **operatore condizionale**) che può regolare il flusso di esecuzione come una condizione. Di seguito si può leggerne la sintassi:

```
variabile = (espr-booleana) ? espr1 : espr2;
```

dove se il valore della `espr-booleana` è `true` si assegna a `variabile` il valore di `espr1`; altrimenti si assegna a `variabile` il valore di `espr2`.

Requisito indispensabile è che il tipo della `variabile` e quello restituito da `espr1` e `espr2` siano compatibili. È escluso il tipo `void`.

L'operatore ternario non può essere considerato un sostituto del costrutto `if`, ma è molto comodo in alcune situazioni. Il seguente codice:

```
String query = "select * from table " +  
(condition != null ? "where " + condition : "");
```

crea una stringa contenente una query SQL e, se la stringa `condition` è diversa da `null`, aggiunge alla query la condizione.

4.3.3 Il costrutto `while`

Questo ciclo permette di iterare uno statement (o un insieme di statement compresi in un blocco di codice) tante volte fino a quando una certa condizione booleana è verificata. La sintassi è la seguente:

```
[inizializzazione;  
while (espr. booleana) {  
    corpo;  
    [aggiornamento iterazione;]  
}
```

Come esempio proponiamo una piccola applicazione che stampa i primi dieci numeri:

```
public class WhileDemo {  
    public static void main(String args[]) {  
        int i = 1;  
        while (i <= 10) {  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

Analizziamo in sequenza le istruzioni che verrebbero eseguite in fase di runtime. Viene in primo

luogo dichiarata ed inizializzata a 1 una variabile intera `i`. Poi inizia il ciclo in cui è esaminato il valore booleano dell'espressione in parentesi. Siccome `i` è uguale ad 1, `i` è anche minore di 10 e la condizione è verificata. Quindi viene eseguito il blocco di codice relativo nel quale prima sarà stampato il valore della variabile `i` (ovvero 1) e poi verrà incrementata la variabile stessa di un'unità. Terminato il blocco di codice, verrà nuovamente testato il valore dell'espressione booleana. Durante questo secondo tentativo la variabile `i` varrà 2. Quindi, anche in questo caso, sarà eseguito di nuovo il blocco di codice. Verrà allora stampato il valore della variabile `i` (ovvero 2) ed incrementata nuovamente di una unità la variabile stessa. Questo ragionamento si ripete fino a quando la variabile `i` non assume il valore 11. Quando ciò accadrà, il blocco di codice non verrà eseguito, dal momento che l'espressione booleana non sarà verificata. Il programma quindi eseguirà le istruzioni successive al blocco di codice (che in questo esempio non ci sono) e quindi terminerà.

4.4 Costrutti di programmazione avanzati

In questo paragrafo ci occuperemo di tutti gli altri costrutti di programmazione che regolano il flusso di un'applicazione.

4.4.1 Il costrutto for

Ecco la sintassi per il `for`, nel caso d'utilizzo di una o più istruzioni da iterare.

Una istruzione:

```
for (inizializzazione; espr. booleana; aggiornamento)  
    istruzione;
```

più istruzioni:

```
for (inizializzazione; espr. booleana; aggiornamento) {  
    istruzione_1;  
    .....;  
    istruzione_n;  
}
```

Il consiglio è sempre di utilizzare comunque i blocchi di codice anche nel caso di istruzione singola.

Presentiamo un esempio che stampa i primi 10 numeri partendo da 10 e terminando a 1:

```
public class ForDemo {  
    public static void main(String args[]) {  
        for (int n = 10; n > 0; n--) {  
            System.out.println(n);  
        }  
    }  
}
```

```
}
```

In questo caso notiamo che la sintassi è più compatta rispetto a quella relativa al `while`. Tra le parentesi tonde relative ad un ciclo `for` dichiariamo addirittura una variabile locale `n` (che smetterà di esistere al termine del ciclo). Potevamo anche dichiararla prima del ciclo, nel caso fosse stata nostra intenzione utilizzarla anche fuori da esso. Per esempio, il seguente codice definisce un ciclo che utilizza una variabile dichiarata esternamente ad esso, e utilizzata nel ciclo e fuori:

```
public void forMethod(int j) {  
    int i = 0;  
    for (i = 1; i < j; ++i) {  
        System.out.println(i);  
    }  
    System.out.println("Numero iterazioni = " + i);  
}
```

La sintassi del ciclo `for` è quindi molto flessibile e compatta. Infatti, se nel `while` utilizziamo le parentesi tonde solo per l'espressione booleana, nel `for` le utilizziamo per inserirci rispettivamente prima l'inizializzazione di una variabile, poi l'espressione booleana ed infine l'aggiornamento che sarà eseguito ad ogni iterazione. Si noti che queste tre istruzioni possono anche essere completamente indipendenti tra loro. Potremmo anche dichiarare più variabili all'interno, più aggiornamenti e, sfruttando operatori condizionali, anche più condizioni. Per esempio il seguente codice è valido:

```
public class For {  
    public static void main(String args[]) {  
        for (int i = 0, j = 10; i < 5 || j > 5; i++, j--) {  
            System.out.println("i=" + i);  
            System.out.println("j=" + j);  
        }  
    }  
}
```

Come è possibile notare, le dichiarazioni vanno separate da virgolette, ed hanno il vincolo di dover essere tutte dello stesso tipo (in questo caso `int`). Anche gli aggiornamenti vanno separati con virgolette, ma non ci sono vincoli in questo caso. Si noti che in questo "settore" del `for` avremmo anche potuto eseguire altre istruzioni, per esempio invocare metodi:

```
for (int i = 0, j = 10; i < 5 || j > 5; i++, j--) {  
    System.out.println("aggiornamento");  
    ...  
}
```

Questo è il ciclo più utilizzato, vista la sua grande semplicità e versatilità. Inoltre, è l'unico ciclo che permette di dichiarare una variabile con visibilità interna al ciclo stesso.

Il ciclo `while` è molto utile quando non si sa quanto a lungo verranno eseguite le istruzioni e soprattutto nei cicli infiniti, dove la sintassi è banale:

```
while (true) {  
    . . .  
}
```

Segue un ciclo `for` infinito:

```
for (;true;) {  
    . . .  
}
```

che è equivalente a:

```
for (;;) {  
    . . .  
}
```

4.4.2 Il costrutto `do`

Nel caso in cui si desideri la certezza che le istruzioni in un ciclo vengano eseguite almeno nella prima iterazione, è possibile utilizzare il ciclo `do`. Di seguito la sintassi:

```
[inizializzazione]  
do {  
    corpo;  
    [aggiornamento iterazione;]  
} while (espr. booleana);
```

In questo caso viene eseguito prima il blocco di codice e poi viene valutata l'espressione booleana (condizione di terminazione) che si trova a destra della parola chiave `while`. Se l'espressione booleana è verificata viene rieseguito il blocco di codice, altrimenti termina. Si noti il punto e virgola situato alla fine del costrutto. L'output del seguente mini-programma:

```
public class DoWhile {  
    public static void main(String args[]) {  
        int i = 10;  
        do {  
            System.out.println(i);  
        } while(i < 10);  
    }  
}
```

è:

4.4.3 Ciclo `for` migliorato

Dalla versione 5 di Java, è stata introdotta una quarta tipologia di ciclo: lo **enhanced for loop**, che in italiano potremmo tradurre con “ciclo for migliorato”. L’aggettivo che però più si addice a questo ciclo non è “migliorato”, bensì “semplificato”. Lo enhanced for infatti non è più potente di un tradizionale ciclo `for` e può sostituirlo solo in alcuni casi. In altri linguaggi il ciclo for migliorato viene chiamato **foreach** (in italiano “per ogni”) e per tale ragione anche in Java si tende ad utilizzare questo nome, sicuramente più agevole da pronunciare e probabilmente più appropriato. La parola chiave `foreach` però, non esiste in Java, e per di più Java 8 ha portato un metodo molto importante che si chiama proprio così nel framework delle Collections (che vedremo più avanti).

Quindi attenzione a non fare confusione! Viene invece riutilizzata la parola chiave `for`. La sintassi è molto semplice e compatta:

```
for (variabile_temporanea : oggetto_iterabile) {  
    //corpo;  
}
```

dove `oggetto_iterabile` è l’array (o un qualsiasi altro oggetto su cui è possibile iterare, come una `Collection` come vedremo più avanti) sui cui elementi si vogliono eseguire le iterazioni. Invece `variabile_temporanea` dichiara una variabile a cui, durante l’esecuzione del ciclo, verrà assegnato il valore dell’i-esimo elemento dell’`oggetto_iterabile` all’i-esima iterazione. In pratica, `variabile_temporanea` rappresenta all’interno del blocco di codice del nuovo ciclo `for` un elemento dell’`oggetto_iterabile`. Quindi, presumibilmente, il `corpo` del costrutto utilizzerà tale variabile. Si noti che non esiste un’espressione booleana per la quale il ciclo deve terminare. Questo è già indicativo del fatto che tale ciclo viene usato soprattutto quando si vuole iterare su tutti gli elementi dell’oggetto iterabile.

Facciamo un esempio:

```
int [] arr = {1,2,3,4,5,6,7,8,9};  
for (int tmp : arr) {  
    System.out.println(tmp);  
}
```

Il precedente frammento di codice stampa a video tutti gli elementi dell’array.

Il ciclo `foreach` ha diversi limiti rispetto al ciclo `for` tradizionale. Per esempio non è possibile eseguire cicli all’indietro, e nemmeno è possibile ciclare su più oggetti contemporaneamente. Infine non è possibile accedere all’indice dell’array dell’elemento corrente. In realtà è sempre possibile dichiarare un contatore all’esterno del ciclo e incrementarlo all’interno, ma a quel punto è forse meglio utilizzare un semplice ciclo `while`.

4.4.4 Il costrutto `switch`

Il costrutto `switch` si presenta come alternativa al costrutto `if`. A differenza di `if` non è possibile utilizzarlo in ogni situazione in cui c’è bisogno di scegliere tra l’esecuzione di parti di codice

diverse. Di seguito presentiamo la sintassi:

```
switch (variabile di test) {  
    case valore_1:  
        istruzione_1;  
        break;  
    case valore_2:  
        istruzione_2;  
        .....;  
    istruzione_k;  
    break;  
    case valore_3:  
    case valore_4: {      //blocco di codice opzionale  
        istruzione_k+1;  
        .....;  
    istruzione_j;  
    }  
    break;  
    [default: {          //clausola default opzionale  
        istruzione_j+1;  
        .....;  
    istruzione_n;  
    }]  
}
```

A seconda del valore intero che assume la variabile di test vengono eseguite determinate espressioni. La variabile di test deve essere o di tipo stringa, o di un tipo di dato compatibile con un intero, ovvero un `byte`, uno `short`, un `char`, oppure direttamente un `int` (ma non un `long`).

Come variabile di test può essere utilizzata anche un'enumerazione o una classe wrapper come `Integer`, `Short`, `Byte` o `Character`. Queste classi sono equiparate ai rispettivi tipi primitivi dalla caratteristica di Java nota come **autoboxing-unboxing**, che permette di interscambiare il tipo primitivo con un oggetto della classe wrapper relativa o viceversa. Tale caratteristica sarà affrontata in dettaglio nel modulo relativo alle librerie fondamentali. Per quanto riguarda le enumerazioni, queste verranno dettagliate nel Modulo 8, dove verrà anche mostrato come si applica al costrutto `switch`.

Inoltre `valore_1...valore_n` devono essere espressioni costanti e diverse tra loro. Si noti che la parola chiave `break` provoca l'immediata uscita dal costrutto. Se dopo aver eseguito tutte le istruzioni che seguono un'istruzione di tipo `case`, non è presente un'istruzione `break`, verranno eseguiti tutti gli statement (istruzioni) che seguono gli altri `case`, sino a quando non si arriverà ad un `break` (questa tecnica è nota come *fallthrough*). Di seguito viene presentato un esempio:

```
public class SwitchStagione {
```

```
public static void main(String args[]) {  
    int mese = 4;  
    String stagione;  
    switch (mese) {  
        case 12:  
        case 1:  
        case 2:  
            stagione = "inverno";  
            break;  
        case 3:  
        case 4:  
        case 5:  
            stagione = "primavera";  
            break; //senza questo break si ha estate  
        case 6:  
        case 7:  
        case 8:  
            stagione = "estate";  
            break;  
        case 9:  
        case 10:  
        case 11:  
            stagione = "autunno";  
            break;  
        default: //la clausola default è opzionale  
            stagione = "non identificabile";  
    }  
    System.out.println("La stagione e' " + stagione);  
}
```

Segue un altro esempio che fa uso di una stringa come variabile di test.

```
public String getTipoGiornoSettimana(String giornoDellaSettimana) {  
    String typeOfDay;  
    switch (giornoDellaSettimana) {  
        case "Monday":  
            typeOfDay = "Inizio settimana";  
            break;  
        case "Tuesday":  
        case "Wednesday":  
            case "Thursday":  
                typeOfDay = "Settimana piena";  
                break;  
        case "Friday":  
            typeOfDay = "Fine settimana lavorativa";  
            break;  
        case "Saturday":
```

```
case "Sunday":  
    typeOfDay = "Weekend";  
default:  
    typeOfDay = "Indefinito!";  
break;  
}  
return typeOfDay;  
}
```

Le stringhe sono case sensitive e bisogna stare attenti. In generale è buona norma evitare il fallthrough, ed usare il costrutto `switch` nella maniera più semplice possibile.

È sempre consigliabile usare una clausola `default` anche quando sembra non essercene bisogno. Infatti a priori non sappiamo se il nostro programma si evolverà ampliando i possibili `case` da definire in uno `switch`. La clausola `default` potrebbe servire sia per gestire i nuovi `case`, con un comportamento standard, sia per scoprire che il costrutto `switch` deve essere modificato per accomodare i nuovi `case`.

Infine è sempre consigliabile mantenere un ordine logico dei vari `case`, per non incorrere in dimenticanze e peggiorare la leggibilità.

Se state utilizzando come editor EJE, potete sfruttare scorciatoie per creare i cinque principali costrutti di programmazione. Potete infatti sfruttare il menu “Inserisci” (o “Insert” se avete scelto la lingua inglese), o le eventuali scorciatoie con la tastiera (CTRL-2, CTRL-3, CTRL-4, CTRL-5, CTRL-6, etc.). In particolare è anche possibile selezionare una parte di codice per poi circondarla con un costrutto.

4.4.5 Due importanti parole chiave: `break` e `continue`

La parola chiave `break` è stata appena presentata come comando capace di fare terminare il costrutto `switch`. Ma `break` è utilizzabile anche per far terminare un qualsiasi ciclo. Il seguente frammento di codice provoca la stampa dei primi dieci numeri interi:

```
int i = 0;  
while (true) //ciclo infinito  
{  
if (i > 10)  
break;  
System.out.println(i);  
i++;  
}
```

Oltre al `break` esiste la parola chiave `continue`, che fa terminare non l'intero ciclo, ma solo l'iterazione corrente.

Il seguente frammento di codice provoca la stampa dei primi dieci numeri, escluso il cinque:

```
int i = 0;
do
{
    i++;
    if (i == 5)
        continue;
    System.out.println(i);
}
while(i <= 10);
```

Sia `break` sia `continue` possono utilizzare etichette (label) per specificare, solo nel caso di cicli annidati, su quale ciclo devono essere applicati. Il seguente frammento di codice stampa, una sola volta, i soliti primi dieci numeri interi:

```
int j = 1;
pippo: //possiamo dare un qualsiasi nome ad una label
while (true)
{
    while (true)
    {
        if (j > 10)
            break pippo;
        System.out.println(j);
        j++;
    }
}
```

Una label ha quindi la seguente sintassi:

```
nomeLabel:
```

Una label può essere posizionata solo prima di un blocco di codice che di solito è definito da un ciclo (come nell'esempio precedente). In realtà potremmo anche posizionare una label al di fuori di un `if` o di un blocco di codice qualsiasi come nel seguente esempio:

```
int j = 1;
labelPerBloccoDiCodice: //possiamo dare un qualsiasi nome ad una label
{
    while (true)
    {
        if (j > 10)
            break labelPerBloccoDiCodice;
        System.out.println(j);
        j++;
    }
}
```

Nell'esempio precedente non potremmo usare il `continue` visto che la label non è relativa a un ciclo. Inoltre il `break` e il `continue` usati con label, devono trovarsi all'interno del blocco di codice etichettato.

Riepilogo

Questo modulo è stato dedicato alla sintassi Java che abbiamo a disposizione per impostare e condizionare il flusso di un programma. Abbiamo descritto (quasi) tutti gli **operatori** supportati da Java, anche quelli meno utilizzati, e i particolari inerenti ad essi. Inoltre abbiamo introdotto tutti i **costrutti** che governano il flusso di esecuzione di un'applicazione dividendoli in costrutti semplici e avanzati. In particolare, abbiamo sottolineato l'importanza della condizione `if` e del ciclo `for`, sicuramente i più utilizzati tra i costrutti. In particolare il ciclo `for` ha una sintassi compatta ed elegante, e soprattutto permette la dichiarazione di variabili con visibilità limitata al ciclo stesso.

In questi primi quattro moduli sono stati affrontati argomenti riguardanti il linguaggio Java. Essi, per quanto non familiari possano risultare al lettore (pensare ai componenti della programmazione come classi e oggetti) rappresentano il nucleo del linguaggio. Il problema è che sino ad ora abbiamo imparato a conoscere questi argomenti ma non ad utilizzarli nella maniera corretta. Ovvero, per quanto il lettore abbia diligentemente studiato, non è probabilmente ancora in grado di sviluppare un programma in maniera corretta. Negli esercizi infatti non è mai stata richiesta l'implementazione di un'applicazione seppur semplice da zero. Per esempio, vi sentite in grado di creare un'applicazione che simuli una rubrica telefonica? Quali classi creereste? Quante classi creereste? Solo iniziare sembra ancora un'impresa, figuriamoci portare a termine l'applicazione. Dal prossimo capitolo, verranno introdotti argomenti riguardanti l'Object Orientation molto più teorici. Questi concetti amplieranno notevolmente gli orizzonti del linguaggio, facendoci toccare con mano vantaggi insperati. Siamo sicuri che il lettore apprenderà agevolmente i concetti che in seguito saranno presentati, ma ciò non basta. Per saper creare un'applicazione da zero bisognerà acquisire esperienza sul campo e un proprio metodo di approccio al problema.

Esercizi modulo 4

Esercizio 4.a)

Scrivere un semplice programma, costituito da un'unica classe, che sfruttando esclusivamente un ciclo infinito, l'operatore modulo, due costrutti `if`, un `break` ed un `continue`, stampi solo i primi cinque numeri pari.

Esercizio 4.b)

Scrivere un'applicazione che stampi i 26 caratteri dell'alfabeto (inglese) con un ciclo.

Esercizio 4.c)

Scrivere una semplice classe che stampi a video la tavola pitagorica.

Suggerimento 1: non sono necessari array.

Suggerimento 2: il metodo `System.out.println()` stampa l'argomento che gli viene passato e poi sposta il cursore alla riga successiva; infatti `println` sta per “print line”. Esiste anche il metodo `System.out.print()` che invece stampa solamente il parametro passatogli.

Suggerimento 3: sfruttare un doppio ciclo innestato.

Esercizio 4.d) Operatori e flusso di esecuzione, Vero o Falso:

1. Gli operatori unari di pre-incremento e post-incremento applicati ad una variabile danno lo stesso risultato, ovvero se abbiamo:

```
int i = 5;
```

sia

```
i++;
```

sia

```
+i;
```

aggiornano il valore di `i` a 6;

2. `d += 1` è equivalente a `d++` dove `d` è una variabile `double`.

3. Se abbiamo:

```
int i = 5;
int j = ++i;
int k = j++;
int h = k--;
boolean flag = ((i != j) && ( (j <= k) || (i <= h)) );
```

`flag` avrà valore `false`.

4. L'istruzione:

```
System.out.println(1 + 2 + "3");
```

stamperà 33.

5. Il costrutto `switch` può in ogni caso sostituire il costrutto `if`.

6. L'operatore ternario può in ogni caso sostituire il costrutto `if`.

7. Il costrutto `for` può in ogni caso sostituire il costrutto `while`.

8. Il costrutto `do` può in ogni caso sostituire il costrutto `while`.

9. Il costrutto `switch` può in ogni caso sostituire il costrutto `while`.

10. I comandi `break` e `continue` possono essere utilizzati nei costrutti `switch`, `for`, `while` e `do` ma non nel costrutto `if`.

Esercizio 4.a)

```
public class TestPari {  
    public static void main(String args[]) {  
        int i = 0;  
        while (true)  
        {  
            i++;  
            if (i > 10)  
                break;  
            if ((i % 2) != 0)  
                continue;  
            System.out.println(i);  
        }  
    }  
}
```

Esercizio 4.b)

```
public class TestArray {  
    public static void main(String args[]) {  
        for (int i = 0; i < 26; ++i) {  
            char c = (char)('a' + i);  
            System.out.println(c);  
        }  
    }  
}
```

Esercizio 4.c)

```
public class Tabelline {  
    public static void main(String args[]) {  
        for (int i = 1; i <= 10; ++i) {  
            for (int j = 1; j <= 10; ++j) {  
                System.out.print(i*j + "\t");  
            }  
            System.out.println();  
        }  
    }  
}
```

Esercizio 4.d) Operatori e flusso di esecuzione, Vero o Falso:

1. **Vero.**

2. **Vero.**

3. **Falso**, la variabile booleana `flag` avrà valore `true`. Le espressioni atomiche valgono

rispettivamente true-false-true, sussistendo le seguenti uguaglianze: i = 6, j = 7, k = 5, h = 6. Infatti (i != j) vale true e inoltre (i <= h) vale true. L'espressione ((j <= k) || (i <= h)) vale true, sussistendo l'operatore OR. Infine l'operatore AND fa sì che la variabile flag valga true.

4. Vero.

5. Falso, switch può testare solo una variabile intera (o compatibile) confrontandone l'uguagliaza con costanti (in realtà dalla versione 5 si possono utilizzare come variabili di test anche le enumerazioni e il tipo Integer, e dalla versione 7 anche le stringhe). Il costrutto if permette di svolgere controlli incrociati sfruttando differenze, operatori booleani etc.

6. Falso, l'operatore ternario è sempre vincolato ad un'assegnazione del risultato ad una variabile. Questo significa che produce sempre un valore da assegnare e utilizzare in qualche modo (per esempio passando un argomento invocando un metodo). Per esempio, se i e j sono due interi, la seguente espressione:

i < j ? i : j;

provocherebbe un errore in compilazione (oltre a non avere senso).

7. Vero.

8. Falso, il do in qualsiasi caso garantisce l'esecuzione della prima iterazione sul codice. Il while potrebbe prescindere da questa soluzione.

9. Falso, lo switch è una condizione non un ciclo.

10. Falso, il continue non si può utilizzare nello switch ma solo nei cicli.

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Conoscere e saper utilizzare i vari operatori (unità 4.1)	<input type="checkbox"/>	
Conoscere e saper utilizzare i costrutti di programmazione semplici (unità 4.2, 4.3)	<input type="checkbox"/>	
Conoscere e saper utilizzare i costrutti di programmazione avanzati (unità 4.2, 4.4)	<input type="checkbox"/>	

Note:

Parte II

Object Orientation

La parte II è interamente dedicata al supporto che Java offre ai paradigmi della programmazione ad oggetti. Questa è forse la parte più importante di questo testo. Infatti dopo aver studiato i primi quattro moduli, anche un lettore neofita della programmazione, dovrebbe già essere in grado di scrivere del codice Java. Ma “smanettare” con un linguaggio di programmazione non è sufficiente, neanche quando si ha una grande capacità. Ma come organizzare un programma partendo da zero? Quante classi bisogna creare? Come si devono chiamare queste classi? E che metodi dovranno avere? Come possiamo creare un programma capace di crescere senza ogni volta ritoccare parti già scritte? A queste e a tante altre domande risponde la teoria dell’Object Orientation. I paradigmi dell’Object Orientation verranno presentati in maniera tale che il lettore impari ad apprezzarne l’utilità in pratica (è consigliato anche lo studio dell’appendice G). In più si è cercato di compiere un’operazione di schematizzazione degli argomenti abbastanza impegnativa, in particolare del polimorfismo. In questo modo si spera di facilitare l’apprendimento. Affronteremo anche il meccanismo della gestione delle eccezioni e della progettazione per contratto mediante le asserzioni.

Consigliamo lo studio di questa sezione a chiunque, anche a chi programma in Java da diverso tempo. Al termine di questa parte il lettore dovrebbe aver appreso le nozioni fondamentali dell’Object Orientation applicabili a Java. Inoltre dovrebbe avere più chiaro come impostare un programma da zero.

Incapsulamento e visibilità

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Comprendere le ragioni della nascita della programmazione ad oggetti (unità 5.1).
- ✓ Saper elencare i paradigmi ed i concetti fondamentali della programmazione ad oggetti (unità 5.2).
- ✓ Saper definire ed utilizzare il concetto di astrazione (unità 5.2).
- ✓ Comprendere l'utilizzo e l'utilità dell'incapsulamento (unità 5.3, 5.4).
- ✓ Comprendere l'utilizzo e l'utilità del reference this (unità 5.3).
- ✓ Capire quando e come usare l'incapsulamento (unità 5.4, 5.7).
- ✓ Comprendere l'utilizzo e l'utilità dei package (unità 5.6, 5.7).
- ✓ Capire e saper utilizzare i modificatori d'accesso (unità 5.8).
- ✓ Capire e saper utilizzare il modificatore static (unità 5.9).

Questo è il primo modulo che si occupa di approfondire il supporto offerto da Java all'Object Orientation. In particolare verranno introdotti i primi paradigmi di questa scienza, come l'incapsulamento, il riuso e l'astrazione. Cercheremo di introdurre il lettore all'argomento, partendo con l'esposizione delle ragioni storiche della nascita dell'Object Orientation. Il modulo contiene anche altri concetti propedeutici allo studio dei prossimi capitoli.

5.1 Breve storia della programmazione ad oggetti

Scrivere un programma significa in qualche modo simulare su un computer concetti e modelli fisici e matematici. Nei suoi primi tempi la programmazione era concepita come una serie di passi lineari. Invece di considerare lo scopo del programma nella sua interezza creandone un modello astratto, si cercava di arrivare alla soluzione del problema superando passaggi intermedi. Questo modello di programmazione orientato ai processi, con il passare del tempo, e con il conseguente aumento delle dimensioni dei programmi, ha manifestato apertamente i suoi difetti. Infatti, aumentando il numero delle variabili e delle interazioni da gestire tra esse, un programmatore in difficoltà aveva a disposizione strumenti come le variabili globali, ed il comando `goto`. In questo modo, agli inizi degli anni Settanta, per una certa programmazione funzionale fu coniato il termine dispregiativo “spaghetti code”, dal momento che i programmi, crescendo in dimensioni, davano sempre più l’idea di assomigliare ad una massa di pasta agrovigliata.

La programmazione orientata agli oggetti nacque storicamente sin dagli anni Sessanta con il linguaggio Simula-67. In realtà non si trattava di un linguaggio orientato agli oggetti puro, ma con

essi furono introdotti fondamentali concetti della programmazione quali le classi e l'ereditarietà. Fu sviluppato nel 1967 da Kristen Nygaard dell'università di Oslo e Ole Johan Dahl del Centro di Calcolo Norvegese e, a dispetto dell'importanza storica, non si può parlare di un vero e proprio successo presso il grande pubblico.

Nei primi anni '70 nacque il linguaggio SmallTalk, sviluppato inizialmente da Alan Kay all'Università dello Utah e successivamente da Adele Goldberg e Daniel Ingalls dello Xerox Park, centro di ricerca di Palo Alto in California. SmallTalk si può considerare un linguaggio ad oggetti puro; introdusse l'incapsulamento e la release SmallTalk-80 ebbe anche un discreto successo negli Stati Uniti. A lungo andare però, sebbene considerato da molti come ideale ambiente di programmazione, rimase confinato (come Simula) nei centri di ricerca universitari di tutto il mondo, considerato come ambiente di studio più che di sviluppo.

L'introduzione nel mondo della programmazione dei concetti di classe e di oggetto, che di fatto rendono i programmi più facilmente gestibili, non provocò quindi immediatamente una rivoluzione nell'informatica. Ciò fu dovuto al fatto che agli inizi degli anni Settanta ottenevano i maggiori successi linguaggi come il C. Un esempio su tutti: il sistema operativo Unix, tutt'oggi ancora utilizzatissimo, nacque proprio in quegli anni, ed il suo kernel (nucleo) era scritto in C.

Negli anni '80 però ci si rese conto dei limiti della programmazione strutturata, il che fu essenzialmente dovuto ad una progressiva evoluzione dell'ingegneria del software, che iniziava a realizzare i programmi con una filosofia incrementale. Per apportare modifiche al software, linguaggi come il C, offrono strumenti come le variabili globali e il comando `goto`, che si possono considerare ad alto rischio. Ecco che allora fu provvidenzialmente introdotta l'estensione del linguaggio C, realizzata da Bjarne Stroustrup, nota con il nome di C++. Questo nuovo linguaggio ha effettivamente rivoluzionato il mondo della programmazione. Fu scelto come linguaggio standard tra tanti linguaggi object oriented dalle grandi corporation (Microsoft, Borland etc.), le quali iniziarono a produrre a loro volta tool di sviluppo che estendevano la programmazione C++.

Essendo un'estensione del C, un qualsiasi programma scritto in C deve poter essere compilato da un compilatore C++. Ciò, anche se ha favorito la migrazione in massa dei programmatore C verso il C++, si è rivelato anche uno dei limiti principali di quest'ultimo. Infatti il programmatore C che migrava a C++ finiva con lo scrivere soprattutto programmi funzionali poco orientati agli oggetti. Da qui l'idea di realizzare un nuovo linguaggio che doveva essere veramente orientato agli oggetti. Java propone uno stile di programmazione che in un certo senso obbliga a programmare correttamente ad oggetti. Inoltre, rispetto al C++, sono stati eliminati tutti gli strumenti ambigui e pericolosi, come ad esempio il `goto`, l'aritmetica dei puntatori e, di fatto, per utilizzare un qualcosa che assomigli ad una variabile globale, ci si deve proprio impegnare!

Possiamo concludere che, se il C++ ha il merito di aver fatto conoscere al grande pubblico la programmazione ad oggetti, Java ha il merito di averla fatta capire!

La programmazione orientata agli oggetti è una scienza, o meglio una filosofia adattabile alla programmazione. Essa si basa su concetti esistenti nel mondo reale, con i quali abbiamo a che fare ogni giorno. È già stato fatto notare al lettore che gli esseri umani posseggono da sempre i concetti di classe e di oggetto. L'astrazione degli oggetti reali in classi fa superare la complessità della realtà. In questo modo possiamo osservare oggetti completamente differenti, riconoscendo in loro caratteristiche e funzionalità che li accomunano, associandoli quindi ad una stessa classe. Per esempio, sebbene completamente diversi, un sassofono ed un pianoforte appartengono entrambi alla

classe degli strumenti musicali. La programmazione ad oggetti inoltre, utilizzando il concetto di incapsulamento, rende i programmi composti da classi che nascondono i dettagli di implementazione dietro ad interfacce pubbliche, le quali permettono la comunicazione tra gli oggetti stessi che fanno parte del sistema. È favorito il riuso di codice già scritto anche grazie a concetti quali l'ereditarietà ed il polimorfismo, che saranno presto presentati al lettore.

C'è da sottolineare che con Java 8 è stato fatto un passo rivoluzionario. Dopo quasi vent'anni sono stati introdotti costrutti tipici della programmazione funzionale per permettere al programmatore di avere maggiore libertà e avere un linguaggio più potente. Tuttavia Java rimane e rimarrà per sempre un linguaggio soprattutto object oriented.

5.2 I paradigmi della programmazione ad oggetti

Ciò che caratterizza un linguaggio orientato agli oggetti è il supporto che esso offre ai cosiddetti **paradigmi della programmazione ad oggetti**:

- Incapsulamento
- Ereditarietà
- Polimorfismo

A differenza di altri linguaggi di programmazione orientati agli oggetti, Java definisce in modo estremamente chiaro i concetti appena accennati. Anche programmatore che si ritengono esperti di altri linguaggi orientati agli oggetti come il C++, studiando Java potrebbero scoprire significati profondi in alcuni concetti che prima si ritenevano chiari.

Nel presente e nei prossimi moduli introdurremo i tre paradigmi in questione. Segnaliamo al lettore che non tutti i testi parlano di tre paradigmi. In effetti si dovrebbero considerare paradigmi della programmazione ad oggetti anche l'astrazione e il riuso (ed altri ancora). Questi due sono spesso considerati secondari rispetto agli altri, non perché meno potenti e utili, ma perché non sono specifici della programmazione orientata agli oggetti. Infatti l'astrazione ed il riuso sono concetti che appartengono anche alla programmazione funzionale.

In realtà i paradigmi fondamentali dell'Object Orientation sono tanti. Ma non sembra questa la sede ideale dove approfondire il discorso.

5.2.1 Astrazione e riuso

L'**astrazione** potrebbe definirsi come **l'arte di sapersi concentrare solo sui dettagli veramente essenziali nella descrizione di un'entità**. In pratica l'astrazione è un concetto chiarissimo a tutti noi, dal momento che lo utilizziamo in ogni istante della nostra vita. Per esempio, mentre state leggendo questo manuale, vi state concentrando sull'apprenderne correttamente i contenuti, senza badare troppo alla forma, ai colori, allo stile e tutti particolari fisici e teorici che compongono la pagina che state visualizzando (o almeno lo speriamo!).

Per formalizzare un discorso altrimenti troppo astratto, potremmo parlare di almeno tre livelli di

astrazione per quanto riguarda la sua implementazione nella programmazione ad oggetti:

- astrazione funzionale
- astrazione dei dati
- astrazione del sistema

Adoperiamo l'astrazione funzionale ogni volta che implementiamo un metodo. Infatti, tramite un metodo, riusciamo a portare all'interno di un'applicazione un concetto dinamico, sinonimo di azione, funzione. Per scrivere un metodo ci dovremmo limitare alla sua implementazione più robusta e chiara possibile. In questo modo avremo la possibilità di invocare quel metodo ottenendo il risultato voluto, senza dover tener presente l'implementazione del metodo stesso.

Lo stesso concetto è valido anche nella programmazione funzionale, grazie alle funzioni.

Adoperiamo l'astrazione dei dati ogni volta che definiamo una classe, raccogliendo in essa solo le caratteristiche e le funzionalità essenziali degli oggetti che essa deve definire nel contesto in cui ci si trova.

Potremmo dire che l'astrazione dei dati “contiene” l'astrazione funzionale.

Adoperiamo l'astrazione del sistema ogni volta che definiamo un'applicazione nei termini delle classi essenziali che devono soddisfare agli scopi dell'applicazione stessa.

Potremmo affermare che l'astrazione del sistema “contiene” l'astrazione dei dati e, per la proprietà transitiva, l'astrazione funzionale.

Il **riuso** è invece da considerarsi una conseguenza dell'astrazione e degli altri paradigmi della programmazione ad oggetti (incapsulamento, ereditarietà e polimorfismo).

Il riuso è un paradigma valido anche per la programmazione funzionale.

5.3 Incapsulamento

L'incapsulamento è la chiave della programmazione orientata agli oggetti. Tramite esso, una classe riesce ad acquisire caratteristiche di robustezza, indipendenza e riusabilità. Inoltre la sua manutenzione risulterà più semplice al programmatore.

Una qualsiasi classe è essenzialmente costituita da dati e metodi. La filosofia dell'incapsulamento è semplice. Essa si basa sull'accesso controllato ai dati mediante metodi che possono prevenirne

l'usura e la non correttezza. A livello di implementazione ciò si traduce semplicemente nel dichiarare privati gli attributi di una classe e quindi inaccessibili fuori dalla classe stessa. A tale scopo introdurremo un nuovo modificatore: `private`.

L'accesso ai dati potrà essere fornito da un'interfaccia pubblica costituita da metodi dichiarati `public` e quindi accessibili da altre classi. In questo modo tali metodi potrebbero ad esempio permettere di realizzare controlli prima di confermare l'accesso ai dati privati.

Se l'incapsulamento è gestito in maniera intelligente le nostre classi potranno essere utilizzate nel modo migliore e più a lungo, giacché le modifiche e le revisioni potranno riguardare solamente parti di codice non visibili all'esterno.

Se volessimo fare un esempio basandoci sulla realtà che ci circonda potremmo prendere in considerazione un telefono. La maggior parte degli utenti, infatti, sa utilizzare il telefono, ma ne ignora il funzionamento interno. Chiunque può alzare la cornetta, comporre un numero telefonico e conversare con un'altra persona, ma pochi conoscono in dettaglio la sequenza dei processi scatenati da queste poche, semplici azioni. Evidentemente, per utilizzare il telefono, non è necessario prendere una laurea in telecomunicazioni: basta conoscere la sua interfaccia pubblica (costituita dalla cornetta e dai tasti), non la sua implementazione interna.

Passiamo ad un esempio di codice. Supponiamo di voler scrivere un'applicazione che utilizza la seguente classe, la quale astrae in maniera semplice il concetto di data:

```
public class Data {  
    public int giorno;  
    public int mese;  
    public int anno;  
}
```

Come può utilizzare la nostra applicazione tale astrazione? Ogni volta che serve un oggetto `Data`, il codice da scrivere sarà simile al seguente:

(Codice 5.1)

```
...  
Data unaData = new Data();  
unaData.giorno = 14;  
unaData.mese = 4;  
unaData.anno = 2004;  
...
```

Dove sta il problema? Non è raro che i valori delle variabili dell'oggetto debbano essere impostati al runtime in maniera dinamica, probabilmente dall'utente. Supponiamo che la nostra applicazione permetta all'utente di inserire la sua data di nascita, magari mediante un'interfaccia grafica. In tal caso il codice da scrivere sarà simile al seguente:

(Codice 5.2)

```
...  
Data unaData = new Data();  
unaData.giorno = interfaccia.dammiGiornoInserito();
```

```
unaData.mese = interfaccia.dammiMeseInserito();
unaData.anno = interfaccia.dammiAnnoInserito();
...
```

dove i metodi `dammiGiornoInserito()`, `dammiMeseInserito()` e `dammiAnnoInserito()` dell'oggetto `interfaccia` restituiscono un intero inserito dall'utente dell'applicazione. Supponiamo che l'utente abbia inserito rispettivamente i valori 32 per il giorno, 13 per il mese e 1800 per l'anno; ecco che i problemi del codice iniziano a risultare evidenti. Come è possibile evitare definitivamente problemi come questo per la classe `Data`? Elenchiamo alcune possibili soluzioni:

1. Si potrebbe limitare la possibilità degli inserimenti sull'interfaccia grafica all'utente. Il problema sarebbe risolto, ma solo nel caso che l'impostazione della data avvenga sempre e comunque tramite l'interfaccia grafica. Inoltre il problema sarà risolto solo per quella particolare funzionalità e quindi non in maniera definitiva. Ma se volessimo riutilizzare (il riuso dovrebbe essere un paradigma fondamentale della programmazione ad oggetti) in un'altra funzione la classe `Data`, senza riutilizzare la stessa interfaccia grafica, saremmo costretti a scrivere nuovamente del codice che gestisce il problema.
2. Potremmo delegare al codice dei metodi `dammiGiornoInserito()`, `dammiMeseInserito()` e `dammiAnnoInserito()` dell'oggetto `interfaccia` i controlli necessari alla giusta impostazione della data. Ma anche in questo caso rimarrebbero tutti i problemi esposti per la soluzione 5.1.
3. Utilizzare l'incapsulamento modificando la classe `Data` nel modo seguente:

(Codice 5.3)

```
public class Data {
private int giorno;
private int mese;
private int anno;

public void setGiorno(int g) {
if (g > 0 && g <= 31) {
giorno = g;
}
else {
System.out.println("Giorno non valido");
}
}

public int getGiorno() {
return giorno;
}

public void setMese(int m) {
if (m > 0 && m <= 12) {
mese = m;
}
}
```

```

else {
System.out.println("Mese non valido");
}
}

public int getMese() {
return mese;
}

public void setAnno(int a) {
anno = a;
}

public int getAnno() {
return anno;
}
}

```

Implementare l'incapsulamento con codice Java consiste il più delle volte nel dichiarare tutti dati privati e fornire alla classe metodi pubblici di tipo “set” e “get” per accedervi rispettivamente in scrittura e lettura.

Questi metodi solitamente (ma non obbligatoriamente) seguono una convenzione che è utilizzata anche nella libreria standard. Se abbiamo una variabile privata dovremmo chiamare questi metodi con la sintassi `setNomeVariabile()` e `getNomeVariabile()`.

Quindi anche se all'inizio potrà sembrare noioso (la seconda versione della classe `Data` è nettamente più estesa della prima) implementare l'incapsulamento non richiede grossa inventiva da parte dello sviluppatore.

Cerchiamo ora di chiarire quali sono i vantaggi. Nel momento in cui abbiamo dichiarato i dati privati, per la definizione del modificatore `private` essi non saranno più accessibili mediante l'operatore dot, a meno che il codice che vuole accedere al dato privato non si trovi nella classe che lo ha dichiarato. Questo implica che il codice 5.1 e il codice 5.2 produrrebbero un errore in compilazione in quanto tenterebbero di assegnare valori a variabili non visibili in quel contesto (classi diverse dalla classe `Data`). I codici 5.1 e 5.2 devono essere rispettivamente sostituiti con i seguenti:

(Codice 5.1.bis)

```

...
Data unaData = new Data();
unaData.setGiorno(14);
unaData.setMese(4);
unaData.setAnno(2004);
...

```

(Codice 5.2.bis)

```
...
Data unaData = new Data();
unaData.setGiorno(interfaccia.dammiGiornoInserito());
unaData.setMese(interfaccia.dammiMeseInserito());
unaData.setAnno(interfaccia.dammiAnnoInserito());
...
```

Ovvero, implementando l'incapsulamento, per sfruttare i dati dell'oggetto `Data`, saremo costretti a utilizzare l'interfaccia pubblica dell'oggetto costituita dai metodi pubblici “set e get”, così come quando vogliamo utilizzare un telefono siamo costretti ad utilizzare l'interfaccia pubblica costituita dai tasti e dalla cornetta. Infatti i metodi “set e get” hanno implementazioni che si trovano internamente alla classe `Data` e quindi possono accedere ai dati privati. Inoltre, nel codice 5.3, si può notare che, per esempio, il metodo `setGiorno()` imposta la variabile `giorno` con il parametro che gli viene passato se risulta compresa tra 1 e 31, altrimenti stampa un messaggio di errore. Quindi, a priori, ogni oggetto `Data` funziona correttamente! Questo implica maggiori opportunità di riuso e robustezza del codice.

Altro immenso vantaggio: il codice è molto più facile da manutenere e si adatta ai cambiamenti. Per esempio, il lettore avrà sicuramente notato che il codice 5.3 risolve relativamente i problemi della classe `Data`. Infatti permetterebbe l'impostazione del giorno al valore 31, anche se la variabile mese vale 2 (Febbraio che ha 28 giorni, ma 29 negli anni bisestili). Bene, possiamo far evolvere la classe `Data`, introducendo tutte le migliorie che vogliamo all'interno del codice 5.3, ma non dovremmo cambiare una riga per i codici 5.1 bis e 5.2 bis! Per esempio, se il metodo `setGiorno()`, viene cambiato nel seguente modo:

```
public void setGiorno(int g) {
    if (g > 0 && g <= 31 && mese != 2) {
        giorno = g;
    }
    else {
        System.out.println("Giorno non valido");
    }
}
```

bisognerà ricompilare solo la classe `Data`, ma i codici 5.1.bis e 5.2.bis rimarranno inalterati!

Al lettore dovrebbe ora risultare chiara l'utilità dei metodi “set” che da ora in poi chiameremo “mutator methods”. Potrebbe però avere ancora qualche riserva sui metodi “get” che da adesso chiameremo “accessor methods”.

A volte ci si riferisce a questi metodi anche come metodi “setter” e “getter”.

Con un paio di esempi dovremmo fugare eventuali dubbi.

Supponiamo di volere verificare dal codice 5.2 bis l'effettivo successo dell'impostazione dei dati

dell'oggetto `unaData`, stampandone i dati a video. Dal momento che la seguente:

```
System.out.println(unaData.giorno);
```

restituirà un errore in compilazione, e:

```
System.out.println(unaData.setGiorno());
```

non ha senso perché il tipo di ritorno del metodo `setGiorno()` è `void`, appare evidente che l'unica soluzione rimane:

```
System.out.println(unaData.getGiorno());
```

Inoltre anche un accessor method potrebbe eseguire controlli come un mutator method. Per esempio, nella seguente classe l'accessor method gestisce l'accesso ad un conto bancario personale, mediante l'inserimento di un codice segreto:

```
public class ContoBancario {  
    private String contoBancario = "5000000 di Euro";  
    private int codice = 1234;  
    private int codiceInserito;  
  
    public void setCodiceInserito(int cod) {  
        codiceInserito = cod;  
    }  
  
    public int getCodiceInserito() {  
        return codiceInserito;  
    }  
  
    public String getContoBancario() {  
        if (codiceInserito == codice) {  
            return contoBancario;  
        }  
        else {  
            return "codice errato!!!";  
        }  
    }  
    . . .  
}
```

5.3.1 Prima osservazione sull'incapsulamento

Sino ad ora abbiamo visto esempi di encapsulamento abbastanza classici, dove nascondevamo all'interno delle classi gli attributi mediante il modificatore `private`. Nulla ci vieta di utilizzare `private`, anche come modificatore di metodi, ottenendo così un **incapsulamento funzionale**. Un metodo privato infatti, potrà essere invocato solo da un metodo definito nella stessa classe, che potrebbe a sua volta essere dichiarato pubblico.

Per esempio la classe `ContoBancario`, definita precedentemente, in un progetto potrebbe evolversi nel seguente modo:

```
public class ContoBancario {  
    . . .  
    public String getContoBancario(int codiceDaTestare) {  
        return controllaCodice(codiceDaTestare);  
    }  
  
    private String controllaCodice(int codiceDaTestare) {  
        if (codiceInserito == codiceDaTestare) {  
            return contoBancario;  
        }  
        else {  
            return "codice errato!!!";  
        }  
    }  
}
```

Ciò favorirebbe il riuso di codice in quanto, introducendo nuovi metodi (come probabilmente accadrà in un progetto che viene manutenuto) questi potrebbero riutilizzare il metodo `controllaCodice()`.

5.3.2 Seconda osservazione sull'incapsulamento

Solitamente si pensa che un membro di una classe dichiarato `private` diventi inaccessibile da altre classi. Questa frase è ragionevole per quanto riguarda l'ambito della compilazione, dove la dichiarazione delle classi è il problema da superare. Ma se ci spostiamo nell'ambito della Java Virtual Machine dove, come abbiamo detto i protagonisti assoluti non sono le classi ma gli oggetti, dobbiamo rivalutare l'affermazione precedente. L'incapsulamento infatti permetterà a due oggetti istanziati dalla stessa classe di accedere in modo pubblico ai rispettivi membri privati.

Consideriamo la seguente classe `Dipendente`:

```
public class Dipendente {  
    private String nome;  
    private int anni; //intendiamo età in anni  
    . . .  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String n) {  
        nome = n;  
    }  
    public int getAnni() {  
        return anni;  
    }  
    public void setAnni(int n) {
```

```
    anni = n;
}
public int getDifferenzaAnni(Dipendente altro) {
return (anni - altro.anni);
}
}
```

Nel metodo `getDifferenzaAnni()` notiamo che è possibile accedere direttamente alla variabile privata `anni` dell'oggetto `altro`, senza dover utilizzare il metodo `getAnni()`.

Il codice precedente è quindi valido per la compilazione, ma il seguente metodo:

```
public int getDifferenzaAnni(Dipendente altro){
return (getAnni() - altro.getAnni());
}
```

favorirebbe sicuramente di più il riuso di codice, e quindi è da considerarsi preferibile. Infatti, il metodo `getAnni()` si potrebbe evolvere introducendo controlli, che conviene richiamare piuttosto che riscrivere.

5.3.3 Il reference `this`

L'esempio precedente potrebbe aver provocato nel lettore qualche dubbio. Sino ad ora avevamo dato per scontato che l'accedere ad una variabile d'istanza all'interno della classe dove è definita fosse un *processo naturale* che non aveva bisogno di reference. Ad esempio, all'interno del metodo `getGiorno()` nella classe `Data` accedevamo alla variabile `giorno` senza referenziarla. Alla luce dell'ultimo esempio e considerando che potrebbero essere istanziati tanti oggetti dalla classe `Data` ci potremmo chiedere: se `giorno` è una variabile d'istanza, a quale istanza appartiene? La risposta a questa domanda è: dipende dall'oggetto corrente, ovvero dall'oggetto su cui è chiamato il metodo `getGiorno()`. Per esempio, in fase d'esecuzione di una certa applicazione potrebbero essere istanziati due particolari oggetti, che supponiamo si chiamino `mioCompleanno` e `tuoCompleanno`. Entrambi questi oggetti hanno una propria variabile `giorno`. Ad un certo punto, all'interno del programma potrebbe presentarsi la seguente istruzione:

```
System.out.println(mioCompleanno.getGiorno());
```

Sarà stampato a video il valore della variabile `giorno` dell'oggetto `mioCompleanno`, ma dal momento che sappiamo come una variabile anche all'interno di una classe potrebbe (e dovrebbe) essere referenziata, dovremmo sforzarci di capire come fa la Java Virtual Machine a scegliere la variabile giusta senza avere a disposizione `reference`!

In realtà si tratta di un'altra iniziativa del compilatore Java. Se il programmatore non referenzia una certa variabile d'istanza, al momento della compilazione il codice sarà modificato dal compilatore stesso, che aggiungerà un `reference` all'oggetto corrente davanti alla variabile. Ma quale `reference` all'oggetto corrente? La classe non può conoscere a priori i `reference` degli oggetti che saranno istanziati da essa in fase di runtime!

Java introduce una parola chiave che per definizione coincide ad un reference all'oggetto corrente: `this` (in italiano “questo”). Il reference `this` viene quindi implicitamente aggiunto nel bytecode compilato, per referenziare ogni variabile d'istanza non esplicitamente referenziata.

Ancora una volta Java cerca di facilitare la vita del programmatore. Infatti in un linguaggio orientato agli oggetti puro, non è permesso non referenziare le variabili d'istanza.

In pratica il metodo `getGiorno()` che avrà a disposizione la JVM dopo la compilazione sarà:

```
public int getGiorno() {  
    return this.giorno; //il this lo aggiunge il compilatore  
}
```

In seguito vedremo altri utilizzi del reference “segreto” `this`.

Anche in questo caso abbiamo notato un altro di quei comportamenti del linguaggio che una volta faceva definire Java “semplice” da imparare. Se non ci siamo posti il problema del referenziare i membri di una classe sino a questo punto, vuol dire che anche questa volta “Java ci ha dato una mano”. Ripetiamo, oggi Java è un linguaggio complicato, ma almeno i suoi creatori hanno creato alcuni meccanismi che ne facilitano l'approccio.

5.3.4 Uso di `this` con variabili

Nel secondo modulo abbiamo distinto le variabili d'istanza dalle variabili locali. La diversità tra i due concetti è tale che il compilatore ci permette di dichiarare una variabile locale (o un parametro di un metodo) ed una variabile d'istanza, aventi lo stesso identificatore, nella stessa classe. Infatti la JVM alloca le variabili locali e le variabili d'istanza in differenti aree di memoria (dette rispettivamente Stack e Heap Memory).

La parola chiave `this` si inserisce in questo discorso nel seguente modo. Abbiamo più volte avuto a che fare con passaggi di parametri in metodi, al fine di inizializzare variabili d'istanza. Sino ad ora, per il parametro passato, siamo stati costretti ad inventare un identificatore differente da quello della variabile d'istanza da inizializzare. Consideriamo la seguente classe:

```
public class Cliente {  
    private String nome, indirizzo, numeroDiTelefono;  
  
    public void setDati(String n, String ind, String num) {  
        nome = n;  
        indirizzo = ind;  
        numeroDiTelefono = num;  
    }  
    //...  
}
```

Notiamo l'utilizzo dell'identificatore `n` per inizializzare `nome`, `num` per `numeroDiTelefono` e `ind` per

indirizzo. Non c'è nulla di sbagliato in questo. Conoscendo però l'esistenza del reference `this`, abbiamo la possibilità di scrivere equivalentemente:

```
public class Cliente {  
    private String nome, indirizzo, numeroDiTelefono;  
  
    public void setDati(String nome, String indirizzo,  
        String numeroDiTelefono) {  
        this.nome = nome;  
        this.indirizzo = indirizzo;  
        this.numeroDiTelefono = numeroDiTelefono;  
    }  
    // . . .  
}
```

Infatti, tramite la parola chiave `this`, specifichiamo che la variabile referenziata appartiene all'istanza. Di conseguenza la variabile non referenziata sarà il parametro del metodo, senza che vi sia ambiguità.

Questo stile di programmazione è in generale considerato preferibile, anzi è uno standard. In questo modo, infatti, non c'è possibilità di confondere le variabili con nomi simili. Nel nostro esempio potrebbe capitare di assegnare il parametro `n` alla variabile d'istanza `numeroDiTelefono` ed il parametro `num` alla variabile `nome`. Potremmo affermare che l'utilizzo di `this` aggiunge chiarezza al nostro codice.

Il lettore noti che se scrivessimo:

```
public class Cliente {  
    private String nome, indirizzo, numeroDiTelefono;  
  
    public void setNumeroDiTelefono(String numeroDiTelefono) {  
        this.numeroDiTelefono = numeroDiTelefono;  
    }  
  
    public void setIndirizzo(String indirizzo) {  
        this.indirizzo = indirizzo;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
    // . . .  
}
```

il compilatore, non trovando riferimenti esplicativi, considererebbe le variabili sempre locali e quindi non otterremmo il risultato desiderato.

5.3.5 Uso di `this` con metodi e costruttori

Per quanto detto, `this` rappresenta l'oggetto corrente e può quindi referenziare le variabili d'istanza. Naturalmente con `this` è possibile anche invocare metodi. Per esempio:

```
public class Cliente {  
  
    private String nome;  
    private String indirizzo;  
    private int numeroDiTelefono;  
    public Cliente(String nome, String indirizzo, String numeroDiTelefono) {  
        this.setNome(nome);  
        this.setIndirizzo(indirizzo);  
        this.setNumeroDiTelefono(numeroDiTelefono);  
    }  
  
    public void setNumeroDiTelefono(String numeroDiTelefono) {  
        this.numeroDiTelefono = numeroDiTelefono;  
    }  
  
    public void setIndirizzo(String indirizzo) {  
        this.indirizzo = indirizzo;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
    // . . .  
}
```

Nel metodo costruttore abbiamo invocato i metodi “set” sfruttando il reference `this`. In questo caso però, se non avessimo anteposto il reference `this` ai metodi “set”, avremmo comunque ottenuto lo stesso risultato, visto che `this` per Java rimane un reference implicito come visto anche per le variabili.

Più interessante è l’uso che si fa solitamente del reference `this` quando abbiamo più costruttori nella stessa classe (overload di costruttori), e da un costruttore se ne vuole chiamare un altro. La sintassi cambia leggermente visto che un costruttore non è un metodo qualsiasi e l’operatore dot non è utilizzabile. Facciamo un esempio:

```
public class Cliente {  
    // . . .  
    public Cliente (String nome, String indirizzo) {  
        this(nome,indirizzo,"sconosciuto");  
    }  
    public Cliente(String nome, String indirizzo, String numeroDiTelefono) {  
        this.setNome(nome);  
        this.setIndirizzo(indirizzo);  
        this.setNumeroDiTelefono(numeroDiTelefono);  
    }  
    // . . .  
}
```

Si noti che abbiamo passato a `this` tre parametri di tipo stringa (non avendo ricevuto in input il numero di telefono abbiamo passato al secondo costruttore un valore di default), e quindi abbiamo chiamato il secondo costruttore (abbiamo riusato codice). In caso di overload di costruttori quindi, la raccomandazione è di far sì che questi si richiamino l'un l'altro per evitare di riscrivere più volte le stesse linee di codice.

5.4 Quando utilizzare l'incapsulamento

Se volessimo essere brevi, dovremmo dire che non ci sono casi in cui è opportuno o meno utilizzare l'incapsulamento. Una qualsiasi classe di una qualsiasi applicazione dovrebbe essere sviluppata utilizzando l'incapsulamento. Anche se all'inizio di un progetto può sembrarci che su determinate classi usufruire dell'incapsulamento sia superfluo, l'esperienza insegna che è preferibile utilizzarlo in ogni situazione. Facciamo un esempio banale. Abbiamo già accennato al fatto che per realizzare un'applicazione a qualsiasi livello (sempre che non sia veramente elementare) bisogna apportare a quest'ultima modifiche incrementalì. Un lettore con un minimo d'esperienza di programmazione non potrà che confermare l'ultima affermazione. Supponiamo di voler scrivere una semplice applicazione che, assegnati due punti, disegni il segmento che li unisce. Supponiamo inoltre che si utilizzi la seguente classe non encapsulata `Punto`, già incontrata nel modulo 2:

```
public class Punto {  
    public int x, y;  
    //...  
}
```

l'applicazione, in un primo momento, istanzierà ed inizializzerà due punti con il seguente frammento di codice:

(Codice 5.4)

```
Punto p1 = new Punto();  
Punto p2 = new Punto();  
p1.x = 5;  
p1.y = 6;  
p2.x = 10;  
p2.y = 20;  
//...
```

Supponiamo che l'evolversi della nostra applicazione renda necessario che i due punti non debbano trovarsi fuori da una certa area piana ben delimitata. Ecco risultare evidente che la soluzione migliore sia implementare l'incapsulamento all'interno della classe `Punto` in questo modo:

```
public class Punto {  
    private int x, y;  
    private final int VALORE_MAXIMO_PER_X=10 ;  
    private final int VALORE_MINIMO_PER_X=-10 ;
```

```

private final int VALORE_MAXIMO_PER_X=10 ;
private final int VALORE_MINIMO_PER_X=-10 ;
public void setX(int a) {
if (a <= VALORE_MAXIMO_PER_X && a >= VALORE_MINIMO_PER_X) {
x = a;
System.out.println("X è OK!");
}
else {
System.out.println("X non valida");
}
}
public void setY(int a){
if (a <= VALORE_MAXIMO_PER_Y && a >= VALORE_MINIMO_PER_Y) {
Y = a;
System.out.println("Y è OK!");
}
else {
System.out.println("Y non valida");
}
}
//. . .
}

```

Purtroppo però, dopo aver apportato queste modifiche alla classe `Punto` saremo costretti a modificare anche il frammento di codice 5.4 dell'applicazione nel modo seguente:

(Codice 5.5)

```

Punto p1 = new Punto();
Punto p2 = new Punto();
p1.setX(5);
p1.setY(6);
p2.setX(10);
p2.setY(20);
. . .

```

Saremmo partiti meglio con la classe `Punto` forzatamente encapsulata in questo modo:

```

public class Punto {
private int x, y;
public void setX(int a) {
x = a;
}
public void setY(int a) {
y = a;
}
. . .
}

```

dal momento che avremmo modificato solo il codice all'interno dei metodi d'accesso e saremmo stati costretti ad utilizzare il codice 5.5 all'interno dell'applicazione che utilizza Punto.

Il codice 5.5 potrebbe essere stato utilizzato in molte altre parti dell'applicazione.

Arriviamo alla conclusione che l'incapsulamento è *prassi ed obbligo* in Java. Un linguaggio orientato agli oggetti puro come lo SmallTalk, infatti, non permetterebbe la dichiarazione di attributi pubblici. Java però vuole essere un linguaggio semplice da apprendere, ed in questo modo non costringe l'aspirante programmatore ad imparare prematuramente un concetto complesso quale l'incapsulamento. In particolare, nei primi tempi, non se ne apprezzerebbe completamente l'utilizzo, dovendo comunque approcciare troppi concetti nuovi contemporaneamente. Tuttavia, non bisognerebbe mai permettere di sacrificare l'incapsulamento per risparmiare qualche secondo di programmazione. Le conseguenze sono ormai note al lettore.

5.5 Come usare l'incapsulamento

Quindi in generale, se parliamo solo del linguaggio e non della filosofia object oriented, l'incapsulamento si riduce semplicemente al **creare classi che dichiarano private le proprie variabili e pubblici i relativi metodi “set” e “get”**. Iniziamo quindi a prendere subito questa buona abitudine, le variabili delle nostre classi vanno sempre dichiarate `private`, e bisogna fornire a corredo i metodi “set” e “get” per renderle utilizzabili all'esterno della classe. Anche se fondamentale, è ovvio che questa attività possa risultare noiosa e soprattutto non è raro sbagliare a scrivere correttamente i metodi “set” e “get”. Basta un lettera minuscola di troppo e si finisce con il perdere tanto tempo. Fortunatamente ci vengono incontro i vari strumenti di sviluppo che permettono di automatizzare la creazione di questi metodi.

Anche EJE consente di creare automaticamente i metodi mutator (set) ed accessor (get) a partire dalla definizione della variabile da encapsulare. Basta aprire il menu “inserisci” e fare clic su “Proprietà JavaBean” (oppure premere CTRL-9). Per Proprietà JavaBean intendiamo una variabile d'istanza encapsulata. Un semplice wizard chiederà di inserire prima il tipo della variabile (per esempio “String”) e poi il nome (per esempio “nome”). Una volta fornite queste due informazioni EJE adempirà al suo compito.

Il nome “Proprietà JavaBean” deriva dal termine dei Java-Beans, una tecnologia che nei primi anni di Java ebbe molto successo, attualmente passata di moda. Il nome JavaBean però è ancora oggi sulla bocca di tutti, anche grazie al “riciclaggio” di tale termine nella tecnologia EJB (Enterprise Java Beans). In inglese vuol dire “chicco di Java” (ricordiamo che Java è il nome di una tipologia di caffè, cfr. appendice A on line).

Ma bisogna per forza dichiarare entrambi i metodi “set” e “get” per ogni variabile? Di solito sì, ma

se per esempio vogliamo limitare l'accesso della variabile d'istanza al di fuori della classe, è possibile anche evitare di dichiarare il metodo "set" (se non vogliamo che possa essere impostata al di fuori della classe), il metodo "get" (nel caso vogliamo impedirne la lettura al di fuori della classe, o anche entrambi i metodi (in questo caso la variabile esisterà e sarà gestita solo all'interno della stessa classe dove è stata definita). Ci sono per esempio casi in cui c'è bisogno di dichiarare classi le cui variabili d'istanza possano essere impostate una sola volta nel momento in cui si istanziano, e il cui valore non cambierà nel tempo. In quel caso basterà non fornire i metodi "set", e sfruttare un costruttore per la prima (e unica) impostazione. Per esempio la seguente classe `Punto` astrae il concetto di punto "fisso":

```
public class PuntoFisso {  
    private int x,y;  
  
    public PuntoFisso (int x, int y){  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX(){  
        return x;  
    }  
  
    public int getY(){  
        return y;  
    }  
}
```

Infatti il valore delle variabili `x` ed `y` non sono modificabili una volta istanziato un oggetto:

```
PuntoFisso puntoFisso = new PuntoFisso(3,5);
```

non esistendo i metodi "set".

5.6 Gestione dei package

Il significato di `public` è chiaro quando applicato a variabili e metodi. Ma come mai sino ad ora abbiamo definito tutte le classi `public`? Per capirlo dobbiamo introdurre meglio il concetto `package`. Abbiamo visto come la libreria standard di Java sia organizzata in `package`. Grazie a questo concetto, il programmatore ha quindi la possibilità di organizzare anche le proprie classi. È molto semplice infatti dichiarare una classe appartenente ad un `package`. La parola chiave `package` permette di specificare, prima della dichiarazione della classe il `package` di appartenenza. Ecco un frammento di codice che ci mostra la sintassi da utilizzare:

```
package programmi.gestioneClienti;  
  
public class AssistenzaClienti {  
    . . . . .
```

L'istruzione package deve essere assolutamente la prima in un file Java.

In questo caso la classe `AssistenzaClienti` apparterrà al package `gestioneClienti` che a sua volta appartiene al package `programmi`. Dichiare la classe `AssistenzaClienti` pubblica, consentirà a tutte le classi che vengono dichiarate anche esternamente al package `programmi.gestioneClienti` di utilizzarla. Se non avessimo anteposto la parola chiave `public` alla classe `AssistenzaClienti`, solo le classi appartenenti allo stesso package avrebbero potuto usare la classe `AssistenzaClienti`.

I package fisicamente sono semplici cartelle (directory). Ciò significa che, dopo aver dichiarato la classe appartenente a questo package, dovremo inserire la classe compilata all'interno di una cartella chiamata `gestioneClienti`, situata a sua volta all'interno di una cartella chiamata `programmi`. Di solito il file sorgente va tenuto separato dalla classe compilata così come schematizzato in Figura 5.1, dove abbiamo idealizzato i file come ovali e le directory come rettangoli.

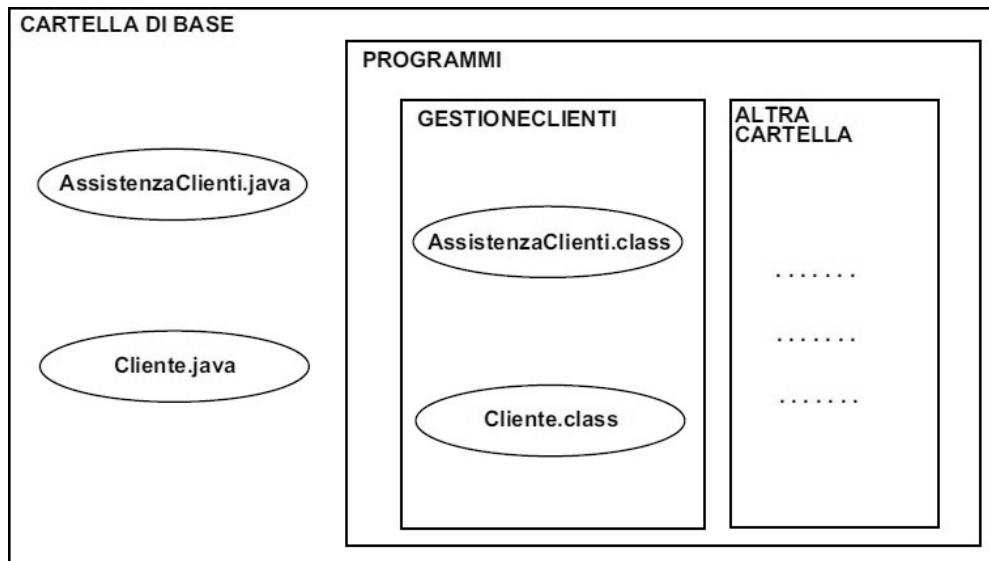


Figura 5.1 – Gestione dei package.

Il Java Development Kit ci permette comunque di realizzare il giusto inserimento dei file nelle cartelle e la relativa creazione automatica delle cartelle stesse, mediante il comando:

```
javac -d . AssistenzaClienti.java
```

Eseguendo il suddetto comando dalla cartella di base (dove è situato il file `AssistenzaClienti.java`), verranno automaticamente create le cartelle `programmi` e `gestioneClienti`, ed il file `AssistenzaClienti.class` collocato nella cartella giusta, senza però

spostare il file sorgente dalla cartella di base. A questo punto, supponendo che il file in questione contenga il metodo `main()`, potremmo eseguirlo solo posizionandoci nella cartella di base ed eseguendo il comando:

```
java programmi.gestioneClienti.AssistenzaClienti
```

infatti, a questo punto, il file è associato ad un package e non potrà essere più chiamato spostandosi nella cartella dove esso è presente utilizzando il comando:

```
java AssistenzaClienti
```

Inoltre bisogna utilizzare un comando di `import` per utilizzare tale classe da un package differente.

Esistono altri problemi collegati all'utilizzo dei package che l'aspirante programmatore potrà incontrare e risolvere con l'aiuto dei messaggi di errore ricevuti dal compilatore. Per esempio potrebbe risultare problematico utilizzare classi appartenenti ad un package, da una cartella non direttamente legata al package stesso. Ciò è risolvibile mediante l'impostazione della variabile di ambiente "classpath" del sistema operativo che dovrebbe puntare alla cartella di base relativa al package in questione. All'argomento classpath è dedicata l'appendice E presente on line.

5.7 Gestione “a mano”

Sebbene sia semplice da spiegare, la gestione dei package può risultare difficoltosa e soprattutto noiosa se “fatta a mano”. Se il lettore è arrivato sino a questo punto utilizzando uno strumento come EJE o il Blocco Note per esercitarsi, vuol dire che è pronto per passare ad un editor più sofisticato. Un qualsiasi tool Java gestisce l'organizzazione dei package in maniera automatica.

Attualmente EJE (versione 3.3) permette di specificare la directory di output dove devono essere generate le classi. Di default però EJE compila le classi nella stessa directory dove si trovano i sorgenti. Ci sono casi in cui per poter compilare correttamente i vostri file bisognerà compilare tutti i file che compongono la vostra applicazione. Non si dovrebbero presentare problemi però se si specifica come directory di output una cartella diversa da quella di default (F12 - Opzioni). In generale tool complessi come Eclipse e Netbeans gestiscono in maniera trasparente i package.

Se il lettore volesse continuare con il Blocco Note (anche se sospettiamo parecchie diserzioni...) ed utilizzare i package, prenda in considerazione il seguente consiglio (altrimenti può anche saltare direttamente la parte restante di questo paragrafo):

- Quando si inizia un progetto è opportuno creare una cartella con il nome del progetto stesso. Per

esempio se il progetto si chiama `Bancomat`, chiamare la cartella `Bancomat`.

- Poi vanno create almeno due sottocartelle, chiamate `src` (dove metteremo tutti i sorgenti) e `classes` (dove metteremo tutti i file compilati). In questo modo divideremo l'ambiente di sviluppo dall'ambiente di distribuzione. Opzionalmente, consigliamo di creare altre cartelle parallele come `docs` (dove mettere tutta la documentazione), `config` (dove mettere i file di configurazione) etc. Per esempio, nella cartella `Bancomat` inserire le cartelle `src`, `classes` etc.
- Ogni volta che una classe deve appartenere ad un package, creare la cartella `package` a mano ed inserire il file sorgente all'interno. Per esempio, se la classe `ContoCorrente` deve appartenere al package `banca`, creare la cartella `banca` all'interno della cartella `src` e inserire il file `ContoCorrente.java` nella cartella `banca`.
- Ogni volta che dobbiamo compilare, bisogna posizionarsi tramite il prompt di DOS nella cartella `classes` (sempre) e da qui eseguire un comando del tipo:

```
javac -d . . .\src\nomepackage\*.java
```

Se volessimo compilare tutte le classi del package `banca` dovremmo eseguire il seguente comando:

```
javac -d . . .\src\banca\*.java
```

Consigliamo al lettore di crearsi un file batch (.bat) per immagazzinare comandi così lunghi o ancora meglio utilizzare un tool come Apache Ant (per informazioni consultare <http://ant.apache.org>).

Questo tipo di organizzazione è proprio quello implementato automaticamente dai tool più importanti come Eclipse e Netbeans.

5.8 Modificatori d'accesso

Sino ad ora abbiamo visto che esistono `private` e `public`, due parole chiavi che rappresentano degli aggettivi con cui andiamo a definire delle caratteristiche per i membri della nostra classe. Questi aggettivi vengono detti modificatori e ne abbiamo già incontrati altri come `final` o `static`. Un modificatore è una parola chiave capace di cambiare il significato di un componente di un'applicazione Java. Come già asserito nel modulo 1, un modifikatore sta ad un componente di un'applicazione Java come un aggettivo sta ad un sostantivo nel linguaggio umano. Ogni modifikatore si applica a uno o più tipologie di componenti Java. Tra queste tipologie sono comprese classi, metodi, variabili d'istanza, variabili locali, attributi di metodi, costruttori ma ne esistono anche altre di cui non abbiamo ancora parlato come le interfacce, le enumerazioni e le annotazioni.

I modificatori protagonisti di questo modulo `public` e `private`, vengono meglio definiti come **modificatori di accesso** perché regolano essenzialmente la visibilità e l'accesso ad un componente Java.

Anche se le sottoclassi saranno l'argomento principale del prossimo modulo dedicato all'ereditarietà, per poter affrontare il discorso sui modificatori d'accesso abbiamo bisogno quantomeno di avere l'idea di cosa sia una sottoclasse. Una sottoclasse è una classe che estende un'altra classe. Per esempio, supponiamo di avere una classe Dipendente e una classe Programmatore che condividono alcune caratteristiche. In particolare supponiamo che entrambe le classi dichiarino alcune variabili stipendio, matricola e annoDiAssunzione, ma che la classe Programmatore dichiari anche un array certificazioni. Queste classi possono essere messe in relazione di ereditarietà tra loro scrivendo semplicemente che la classe Programmatore "estende" la classe Dipendente. Così facendo, la classe Programmatore non deve ridichiarare tutte le caratteristiche in comune con la classe Dipendente (perché saranno ereditate).

I modificatori d'accesso di Java sono i seguenti:

- **public**: può essere utilizzato sia per un membro (attributo o metodo) di una classe, sia per una classe stessa. Sappiamo oramai bene che un membro dichiarato pubblico sarà accessibile da una qualsiasi classe situata in qualsiasi package e che per l'incapsulamento le variabili d'istanza non dovrebbero mai essere dichiarate pubbliche. Una classe dichiarata pubblica sarà anch'essa visibile da un qualsiasi package, come abbiamo già detto.
- **protected**: questo modificatore definisce per un membro il grado più accessibile dopo quello definito da `public`. Un membro protetto sarà infatti accessibile all'interno dello stesso package ed in tutte le sottoclassi della classe in cui è definito, anche se non appartenenti allo stesso package. È invece un errore comune pensare che `protected` renda accessibili i membri di una classe alle sole sottoclassi, e non alle classi dello stesso package. Non è possibile dichiarare una classe `protected` perché non ha senso.
- **Nessun modificatore**: possiamo evitare di usare modificatori sia relativamente ad un membro (attributo o metodo) di una classe sia relativamente ad una classe stessa. Se non anteponiamo modificatori d'accesso ad un membro di una classe, esso sarà accessibile solo da classi appartenenti al package dove è definito. Se invece dichiariamo una classe appartenente ad un package senza anteporre alla sua definizione il modificatore `public`, la classe stessa sarà visibile solo dalle classi appartenenti allo stesso package. In pratica dovremmo parlare di "visibilità di package".
- **private**: questo modificatore restringe la visibilità di un membro di una classe alla classe stessa (ma bisogna tener conto di quanto osservato nel paragrafo 5.3.2 *Seconda osservazione sull'incapsulamento*).

Il tutto è riassunto nella seguente tabella riguardante i modificatori di accesso e la relativa visibilità (solo per i membri di una classe):

MODIFICATORE	STESSA CLASSE	STESO PACKAGE	SOTTOCLASSE	OVUNQUE
public	SI	SI	SI	SI
protected	SI	SI	SI	NO
nessun modificatore	SI	SI	NO	NO
private	SI	NO	NO	NO

5.9 Il modificatore static

static è forse il più potente modificatore di Java. Forse anche troppo! Con static la programmazione ad oggetti trova un punto di incontro con quella strutturata ed il suo uso deve essere quindi limitato a situazioni di reale e concreta utilità. Potremmo tradurre il termine static con “condiviso da tutte le istanze della classe”, oppure “della classe”. Quindi le regole dell’incapsulamento quando parliamo di questo modificatore sono alterate. È possibile encapsulare una variabile statica, ma quella variabile verrà comunque condivisa da tutti gli oggetti istanziati dalla stessa classe. In generale quindi l’incapsulamento non si applica a variabili statiche (che come vedremo tra poco dovrebbero essere usate raramente).

Per quanto detto, un membro statico ha la caratteristica di poter essere utilizzato mediante una sintassi del tipo:

```
NomeClasse.nomeMembro
```

in luogo di:

```
nomeOggetto.nomeMembro
```

Possiamo usare static per marcare i metodi e gli attributi di una classe (quindi i membri di una classe) e come vedremo tra poco anche gli inizializzatori. Non è possibile invece applicare questo modificatore a variabili locali, parametri di metodi, costruttori o classi. Anche senza istanziare la classe, l’utilizzo di un membro statico provocherà il caricamento in memoria della classe contenente il membro in questione, che quindi, condividerà il ciclo di vita con quello della classe.

Si possono anteporre alla dichiarazione di un componente di un’applicazione Java anche più modificatori alla volta, senza tener conto dell’ordine in cui vengono anteposti. Una variabile dichiarata static e public per esempio, avrà quindi le stesse proprietà di una dichiarata public e static.

5.9.1 Metodi statici

Un esempio di metodo statico è il metodo `sqrt()` (che sta per “square root” ovvero “radice quadrata”) della classe `Math` (package `java.lang`), che viene chiamato tramite la sintassi:

```
Math.sqrt(numero)
```

Math è quindi il nome della classe e non il nome di un'istanza di quella classe. La ragione per cui la classe `Math` dichiara tutti i suoi metodi statici è facilmente comprensibile. Infatti, se istanziassimo due oggetti differenti dalla classe `Math`, `ogg1` e `ogg2`, i due comandi:

```
ogg1.sqrt(4);
```

e

```
ogg2.sqrt(4);
```

produrrebbero esattamente lo stesso risultato (2). Effettivamente non ha senso istanziare due oggetti di tipo “matematica”.

Un metodo dichiarato `static` e `public` può considerarsi una sorta di funzione (non a caso nella classe `Math` la funzione radice quadrata è stata resa pubblica e statica) e perciò questo tipo di approccio ai metodi dovrebbe essere evitato nella maggior parte dei casi. La creazione di un metodo statico dovrebbe essere un evento raro, limitato solo ai casi in cui l'esecuzione del metodo non dipenda dalle caratteristiche dell'oggetto su cui viene chiamato il metodo.

5.9.2 Variabili statiche (di classe)

Una variabile statica, essendo condivisa da tutte le istanze della classe, assumerà lo stesso valore per ogni oggetto di una classe.

Di seguito viene presentato un esempio:

```
public class ClasseDiEsempio {  
    public static int a = 0;  
}  
  
public class ClasseDiEsempioPrincipale {  
    public static void main (String args[]) {  
        System.out.println("a = " + ClasseDiEsempio.a);  
        ClasseDiEsempio ogg1 = new ClasseDiEsempio();  
        ClasseDiEsempio ogg2 = new ClasseDiEsempio();  
        ogg1.a = 10;  
        System.out.println("ogg1.a = " + ogg1.a);  
        System.out.println("ogg2.a = " + ogg2.a);  
        ogg2.a = 20;  
        System.out.println("ogg1.a = " + ogg1.a);  
        System.out.println("ogg2.a = " + ogg2.a);  
    }  
}
```

L'output di questo semplice programma sarà:

```
a = 0
```

```
ogg1.a = 10  
ogg2.a = 10  
ogg1.a = 20  
ogg2.a = 20
```

Come si può notare, se un'istanza modifica la variabile statica, essa risulterà modificata anche relativamente all'altra istanza. Infatti essa è condivisa dalle due istanze ed in realtà risiede nella classe. Una variabile di questo tipo potrebbe ad esempio essere utile per contare il numero di oggetti istanziati da una classe (per esempio incrementandola in un costruttore). Per esempio:

```
public class Counter {  
    private static int counter = 0;  
    private int number;  
    public Counter() {  
        counter++;  
        setNumber(counter);  
    }  
    public void setNumber(int number) {  
        this.number = number;  
    }  
    public int getNumber() {  
        return number;  
    }  
}
```

Si noti come la variabile d'istanza `number` venga valorizzata nel costruttore di questa classe con il valore della variabile statica `counter`.

A differenza di una variabile d'istanza ordinaria, una variabile statica non viene inizializzata al valore nullo del suo tipo al momento dell'istanza dell'oggetto: non è una variabile d'istanza.

Questo significa che `number` rappresenterà proprio il numero seriale di ogni oggetto istanziato. Per esempio, dopo questa istruzione:

```
Counter c1 = new Counter();
```

La variabile statica `counter` varrà 1 e la variabile `c1.number` varrà sempre 1.

Se poi istanziamo un altro oggetto `Counter`:

```
Counter c2 = new Counter();
```

allora la variabile statica `counter` varrà 2. Infatti, essendo condivisa da tutte le istanze della classe, non viene riazzerrata ad ogni istanza. Invece la variabile `c1.number` varrà sempre 1, mentre la variabile `c2.number` varrà 2.

Il modificatore `static` quindi prescinde dal concetto di oggetto e lega strettamente le variabili al concetto di classe, che a sua volta si innalza a qualcosa più di un semplice mezzo per definire oggetti. Per questo motivo a volte ci si riferisce alle variabili d'istanza statiche come “variabili di classe”.

Una variabile dichiarata `static` e `public` può considerarsi una sorta di “variabile globale”. Le variabili globali rappresentano una scorciatoia per i programmatori, ma sono difficili da gestire e per questo sono a ragione considerate esempio di cattiva progettazione. Questo tipo di approccio alle variabili quindi va accuratamente evitato. È però utile a volte utilizzare costanti globali, definite con `public, static` e `final`. Per esempio la classe `Math` definisce due costanti statiche pubbliche: `PI` ed `E` (cfr. documentazione ufficiale).

Un metodo statico non può utilizzare variabili d'istanza senza referenziarle, ma solo variabili statiche (e variabili locali). Infatti, un metodo statico non appartiene a nessuna istanza in particolare, e quindi non potrebbe “scegliere” una variabile d'istanza di una istanza particolare senza referenziarla esplicitamente. Per esempio la seguente classe:

```
public class StaticMethod {  
    private int variabileDiIstanza;  
    private static int variabileDiClasse;  
    public static void main(String args[]) {  
        System.out.println(variabileDiIstanza);  
    }  
}
```

produrrà il seguente errore in compilazione:

```
non-static variable variabileDiIstanza cannot be referenced from a static  
context  
System.out.println(variabileDiIstanza);  
^  
1 error
```

Se invece stampassimo la variabile `variabileDiClasse` il problema non si porrebbe.

5.9.3 Inizializzatori statici ed inizializzatori d'istanza

Il modificatore `static` può anche essere utilizzato per marcare un semplice blocco di codice definito all'interno di una classe (ma al di fuori di un metodo). Stiamo parlando di un nuovo componente (per la verità usato raramente) che viene chiamato **inizializzatore statico**. In pratica un blocco statico definito all'interno di una classe avrà la caratteristica di essere chiamato al momento del caricamento in memoria della classe stessa, addirittura prima di un eventuale costruttore. La sintassi è semplice come mostrato nel seguente esempio:

```
public class EsempioStatico {
```

```
private static int a = 10;
public EsempioStatico(){
    a += 10;
}
static {
    System.out.println("valore statico = " + a);
}
```

Anche questo blocco, come nel caso dei metodi statici, potrà utilizzare variabili definite fuori da esso se e solo se dichiarate statiche.

Supponiamo di istanziare un oggetto di questa classe mediante la seguente sintassi:

```
EsempioStatico ogg = new EsempioStatico();
```

Questo frammento di codice produrrà il seguente output:

```
valore statico = 10
```

Infatti, quando si istanzia un oggetto da una classe, questa deve essere prima caricata in memoria. È in questa fase di caricamento che viene eseguito il blocco statico, e di conseguenza stampato il messaggio di output. Successivamente viene chiamato il costruttore che incrementerà il valore statico.

È possibile inserire in una classe anche più di un inizializzatore statico. Questi verranno eseguiti in maniera sequenziale “dall’alto in basso”.

L’uso di un inizializzatore statico può effettivamente essere considerato sporadico, e legato a soluzioni di progettazione avanzate.

Esiste anche un inizializzatore non statico. Si chiama **inizializzatore d’istanza (instance initializer o object initializer)** e si implementa includendo codice in un blocco di parentesi graffe all’interno di una classe. La sua caratteristica è l’essere eseguito quando viene istanziato un oggetto, prima del costruttore. Per esempio, se istanziassimo la seguente classe:

```
public class InstanceInitializer {
    public InstanceInitializer() {
        System.out.println("Costruttore");
    }
    {
        System.out.println("Inizializzatore");
    }
}
```

l'output risultante sarebbe il seguente:

Inizializzatore
Costruttore

Anche per l'inizializzatore d'istanza, le situazioni in cui è necessario utilizzarlo sono molto rare.

È possibile inserire nella stessa classe più inizializzatori statici e d'istanza. Gli inizializzatori verranno eseguiti in maniera sequenziale a seconda di come sono stati ordinati all'interno del file, ma quelli statici avranno sempre la precedenza.

5.9.4 import statici

Nella versione 5 di Java, furono introdotti (insieme a numerose altre novità) anche i cosiddetti **import statici**. Con il comando `import` siamo soliti importare nei nostri file classi ed interfacce direttamente da package esterni. In pratica, possiamo importare all'interno dei nostri file i nomi delle classi e delle interfacce, in modo tale che si possano poi usare senza dover specificare l'intero “fully qualified name” (nome completo di package). Per esempio, volendo utilizzare la classe `DOMSource` del package `javax.xml.transform.dom`, abbiamo due possibilità:

- Importiamo la classe con un'istruzione come la seguente:

```
import javax.xml.transform.dom.DOMSource;
```

per poi sfruttarla nel nostro file sorgente utilizzandone solo il nome. Per esempio:

```
DOMSource source = new DOMSource();
```

- Oppure possiamo scrivere il nome completo di package ogni volta che la utilizziamo. Per esempio:

```
javax.xml.transform.dom.DOMSource source = new javax.xml.transform.dom.DOMSource();
```

Come si può intuire la prima soluzione è molto più conveniente.

In alcuni casi però, potremmo desiderare di importare nel file solo ciò che è dichiarato statico all'interno di una certa classe, e non la classe stessa. Per esempio, sapendo che la classe `Math` contiene solo metodi e costanti statiche, potremmo voler importare, piuttosto che la classe `Math`, solo i suoi membri statici. Segue la sintassi per realizzare quanto detto:

```
import static java.lang.Math.*;
```

In questo caso abbiamo importato tutti i membri statici all'interno del file. Quindi sarà lecito scrivere, in luogo di:

```
double d = Math.sqrt(4);
```

direttamente:

```
double d = sqrt(4);
```

senza anteporre il nome della classe al metodo statico.

È anche possibile importare staticamente solo alcuni membri specifici, per esempio:

```
import static java.lang.Math.PI;
import static java.lang.Math.random;
import static java.sql.DriverManager.getConnection;
import static java.lang.System.out;
```

Si noti come l'`import` di nomi di metodi statici non specifichi le parentesi con i relativi argomenti.

L'utilizzo di `import static` non è sempre vantaggioso e in generale ne sconsigliamo l'utilizzo. Tutte le considerazioni e i consigli di utilizzo di questa caratteristica di Java sono trattati nell'appendice F on line.

5.9.5 Quando usare `static`

Il lettore avrà sicuramente notato che l'uso di `static` è stato scoraggiato praticamente in ogni paragrafo. In effetti se volessimo atteggiarci a puristi dell'Object Orientation, dovremmo affermare che `static` è semplicemente una parola da non usare nella programmazione Java. In teoria se ne potrebbe fare tranquillamente a meno. Ma `static` non è l'impersonificazione del male! Sicuramente bisogna evitare variabili globali, creare funzioni al posto di metodi, abusare di `import static` senza motivo. Ci sono casi in cui si possono risolvere problemi di programmazione in maniera elegante e funzionale. Per esempio nel caso del contatore di istanze visto nel paragrafo 5.9.2, una variabile statica ci ha permesso di realizzare il nostro obiettivo con il minimo sforzo. Le costanti statiche (attributi dichiarati `public`, `final` e `static`) sono usatissime. Abbiamo anche visto che il metodo statico `sqrt()` della classe `Math` è correttamente definito (perché istanziare un oggetto della classe `Math` senza che ce ne sia bisogno?). Per quanto riguarda l'incapsulamento (argomento principale di questo modulo) è possibile creare attributi statici e incapsularli con metodi "set" e "get" (sempre statici). Consideriamo il seguente codice:

```
public class IncapsulamentoStatico {
private static int attributoStatico = 0;
public static void setAttributoStatico(int attributoStatico) {
this.attributoStatico = attributoStatico;
}
public static int getAttributoStatico() {
return attributoStatico;
}
}
```

Il codice non compilerà! Di seguito l'output della compilazione:

```
D:\java8\Codice\modulo_05\esempi\IncapsulamentoStatico.java:4:  
error: non-static variable this cannot be referenced from a static context  
this.attributoStatico = attributoStatico;  
^  
D:\java8\Codice\modulo_05\esempi\IncapsulamentoStatico.java:4:  
warning: [static] static variable should be qualified by type name,  
IncapsulamentoStatico, instead of by an expression  
this.attributoStatico = attributoStatico;  
^  
1 error  
1 warning
```

Questo perché il reference `this` abbiamo visto che si riferisce all'oggetto corrente e quindi non può essere usato nel contesto di un metodo statico, e questo viene segnalato come errore. Inoltre viene segnalato un warning (non è un errore ma solo un avvertimento del compilatore) che ci avverte che `this` si riferisce ad un oggetto, ma la variabile `attributoStatico` appartiene alla classe `IncapsulamentoStatico` e sarebbe preferibile usare la classe per referenziare l'attributo.

Il codice quindi compilerà se il metodo “set” verrà modificato nel seguente modo:

```
public static void setAttributoStatico(int attributoStatico) {  
    IncapsulamentoStatico.attributoStatico = attributoStatico;  
}
```

Di fatto quindi è possibile sfruttare l'incapsulamento per variabili statiche, così però non incapsuleremo oggetti, ma classi. Un'implementazione dell'incapsulamento statico molto famosa (e molto utile) è il design pattern **Singleton**.

5.9.6 Design Pattern Singleton

I Design Pattern sono soluzioni progettuali standard applicabili a contesti diversi. Solitamente se durante la fase di progettazione di un'applicazione si presenta un problema progettuale, la maggior parte delle volte non c'è bisogno di reinventare l'acqua calda. Una soluzione standard esiste già (è stata trovata e formalizzata in passato da progettisti esperti) ed è applicabile, basta solo conoscerla ed implementarla nel nostro contesto (ma non è sempre semplice!).

On line troverete l'appendice D dedicata all'introduzione dei Design Pattern.

Un ottimo esempio di Design Pattern è il **Singleton**. Se abbiamo la necessità che una classe debba essere istanziata una sola volta, e che quindi tutti gli utilizzatori di questa classe utilizzino sempre la stessa istanza, la soluzione consiste nell'utilizzo del pattern Singleton. L'implementazione classica del pattern Singleton, consiste nel creare una classe con:

1. un costruttore privato. In questo modo il costruttore che è necessario chiamare per istanziare un

oggetto da questa classe è utilizzabile solo all'interno della classe stessa.

2. Un variabile privata e statica dello stesso tipo della classe (di solito chiamato `instance` ovvero “istanza”).
3. Un metodo statico pubblico (solitamente chiamato `getInstance()`) che definisce una semplice logica per restituire sempre la stessa istanza (unica) della classe stessa.

Ecco un esempio di implementazione del pattern Singleton:

```
public class SingletonExample {  
    private static SingletonExample instance;  
  
    private SingletonExample () {  
    }  
  
    public static SingletonExample getInstance() {  
        if (instance == null) {  
            instance = new SingletonExample();  
        }  
        return instance;  
    }  
}
```

Si noti come il metodo `getInstance()` essendo interno alla classe possa utilizzare il metodo costruttore (anche se privato) per istanziare la classe stessa. Per la logica che definisce verrà creato un oggetto solo la prima volta che verrà chiamato questo metodo ed assegnato all'attributo statico `instance`. Dalla seconda chiamata in poi questo metodo restituirà sempre la stessa istanza.

Quindi per ottenere l'unica istanza della classe `SingletonExample` le altre classi dovranno usare questa sintassi:

```
SingletonExample unicalistanza = SingletonExample.getInstance();
```

L'utilizzo dei pattern permette anche ai programmatore di usare una terminologia semplificata per parlare del codice. I programmatore spesso comunicano utilizzando la terminologia dei pattern (per esempio “... questa classe è un Singleton ...”). Inoltre alcuni IDE automatizzano le implementazioni dei design pattern in maniera automatica.

Anche EJE permette di implementare automaticamente il pattern Singleton. Dopo aver creato una classe semplicemente fare clic sul menu Insert ➔ Singleton o usare la combinazione di tasti CTRL-F9.

Riepilogo

In questo modulo è stato introdotto il supporto che offre Java all'Object Orientation. Dopo una panoramica storica sono stati elencati alcuni fondamentali paradigmi, ovvero **astrazione, riuso,**

incapsulamento, ereditarietà e polimorfismo. Sono stati formalizzati il concetto di astrazione e quello di riuso. Anche essendo paradigmi fondamentali della programmazione orientata agli oggetti, l'astrazione e il riuso sono paradigmi validi anche per la programmazione strutturata. Per questa ragione sono spesso considerati paradigmi secondari, ma il lettore dovrebbe tenere ben presente che il loro utilizzo è assolutamente cruciale per programmare in Java, come cercheremo di dimostrare in questo manuale.

Sono stati poi esposti al lettore i vantaggi dell'incapsulamento, ovvero maggiore manutenibilità, robustezza e riusabilità. Scrivere codice per implementare encapsulamento è molto semplice, tanto che la maggior parte degli strumenti di sviluppo supporta un modo standard per encapsulare gli attributi di una classe. Inoltre è anche semplice capire quando applicare questo concetto: sempre!

Abbiamo anche introdotto il reference `this` anche se per ora non sembra una parola chiave indispensabile per programmare. La sua introduzione però, è stata utile per chiarire alcuni punti oscuri che potevano (anche inconsciamente) rappresentare un ostacolo alla completa comprensione degli argomenti.

Sono stati definiti anche il concetto di **package** e l'eventuale gestione “a mano” che potremmo usare se lavorassimo da riga di comando senza un IDE. Nonostante nella programmazione avanzata i package siano utilizzati sempre, abbiamo sottolineato quanto questo possa inizialmente influire sulla curva d'apprendimento di un lettore che si sta solo avvicinando alla programmazione Java, e voglia usare tutto. Tuttavia, usando un IDE come Eclipse o Netbeans, l'utilizzo dei package è praticamente trasparente. Sono stati poi spiegati i significati dei **modificatori d'accesso** e la loro applicabilità ai componenti del linguaggio Java. Infine è stato spiegato il modificatore fondamentale `static` e la sua applicabilità a metodi e variabili (che vengono chiamate **variabili di classe**). Ne abbiamo approfittato per introdurre anche un nuovo tipo di componente: gli **inizializzatori** (statici e di oggetto). Abbiamo anche mostrato l'esistenza degli **import statici** e ne abbiamo per ora sconsigliato l'utilizzo. Abbiamo anche dimostrato come questo modificatore sia adatto a risolvere problemi particolari, e che non è uno strumento di cui abusare. Ne abbiamo approfittato per introdurre i Design Pattern e in particolare il pattern **Singleton**.

Esercizi modulo 5

Esercizio 5.a) Object Orientation in generale (teoria), Vero o Falso:

1. L'Object Orientation esiste solo da pochi anni.
2. Java è un linguaggio object oriented non puro, SmallTalk è un linguaggio object oriented puro.
3. Tutti i linguaggi orientati agli oggetti supportano allo stesso modo i paradigmi object oriented.
Si può dire che un linguaggio è object oriented se supporta encapsulamento, ereditarietà e polimorfismo; infatti altri paradigmi come l'astrazione e il riuso appartengono anche alla programmazione funzionale.
4. Applicare l'astrazione significa concentrarsi solo sulle caratteristiche importanti dell'entità da astrarre.
5. La realtà che ci circonda è fonte d'ispirazione per la filosofia object oriented.

6. L'incapsulamento ci aiuta ad interagire con gli oggetti, l'astrazione ci aiuta ad interagire con le classi.
7. Il riuso è favorito dall'implementazione degli altri paradigmi object oriented.
8. L'ereditarietà permette al programmatore di gestire in maniera collettiva più classi.
9. L'incapsulamento divide gli oggetti in due parti separate: l'interfaccia pubblica e l'implementazione interna.
10. Per l'utilizzo dell'oggetto basta conoscere l'implementazione interna e non bisogna conoscere l'interfaccia pubblica.

Esercizio 5.b) Incapsulare e completare le seguenti classi:

```
public class Pilota {  
    public String nome;  
  
    public Pilota(String nome) {  
        // impostare il nome  
    }  
}  
  
public class Auto {  
    public String scuderia;  
    public Pilota pilota;  
  
    public Auto (String scuderia, Pilota pilota) {  
        // impostare scuderia e pilota  
    }  
    public String dammiDettagli() {  
        // restituire una stringa descrittiva dell'oggetto  
    }  
}
```

Tenere presente che le classi `Auto` e `Pilota` devono poi essere utilizzate dalle seguenti classi:

```
public class TestGara {  
    public static void main(String args[]) {  
        Gara imola = new Gara("GP di Imola");  
        imola.corriGara();  
        String risultato = imola.getRisultato();  
        System.out.println(risultato);  
    }  
}  
  
public class Gara {  
    private String nome;  
    private String risultato;  
    private Auto griglia [];
```

```

public Gara(String nome) {
setNome(nome);
setRisultato("Corsa non terminata");
creaGrigliaDiPartenza();
}

public void creaGrigliaDiPartenza(){
Pilota uno = new Pilota("Pippo");
Pilota due = new Pilota("Pluto");
Pilota tre = new Pilota("Topolino");
Pilota quattro = new Pilota("Paperino");
    Auto autoNumeroUno = new Auto("Ferrari", uno);
Auto autoNumeroDue = new Auto("Renault", due);
Auto autoNumeroTre = new Auto("BMW", tre);
Auto autoNumeroQuattro = new Auto("Mercedes", quattro);
griglia = new Auto[4];
griglia[0] = autoNumeroUno;
griglia[1] = autoNumeroDue;
griglia[2] = autoNumeroTre;
griglia[3] = autoNumeroQuattro;
}

public void corriGara() {
int numeroVincente = (int)(Math.random()*4);
Auto vincitore = griglia[numeroVincente];
String risultato = vincitore.dammiDettagli();
setRisultato(risultato);
}

public void setRisultato(String vincitore) {
this.risultato = "Il vincitore di " + this.getNome() + ": " + vincitore;
}

public String getRisultato() {
return risultato;
}

public void setNome(String nome) {
this.nome = nome;
}

public String getNome() {
return nome;
}
}

```

Analisi dell'esercizio

La classe `TestGara` contiene il metodo `main()` e quindi determina il flusso di esecuzione dell'applicazione. È molto leggibile: si istanzia un oggetto `gara` e lo si chiama “GP di Imola”, si fa correre la corsa, si richiede il risultato e lo si stampa a video.

La classe `Gara` invece contiene pochi e semplici metodi e tre variabili d'istanza: `nome` (il nome della gara), `risultato` (una stringa che contiene il nome del vincitore della gara se è stata corsa) e `griglia` (un array di oggetti `Auto` che partecipano alla gara).

Il costruttore prende in input una stringa con il nome della gara che viene opportunamente impostato. Inoltre il valore della stringa `risultato` è impostata a “Corsa non terminata”. Infine è chiamato il metodo `creaGrigliaDiPartenza()`.

Il metodo `creaGrigliaDiPartenza()` istanzia quattro oggetti `Pilota` assegnando loro dei nomi. Poi istanzia quattro oggetti `Auto` assegnando loro i nomi delle scuderie ed i relativi piloti. Infine istanzia ed inizializza l'array `griglia` con le auto appena create.

Una gara dopo essere stata istanziata è pronta per essere corsa.

Il metodo `corriGara()` contiene codice che va analizzato con più attenzione. Nella prima riga, infatti, viene chiamato il metodo `random()` della classe `Math` (appartenente al package `java.lang` che viene importato automaticamente). La classe `Math` astrae il concetto di “matematica” e sarà descritta più avanti in questo libro. Essa contiene metodi che astraggono tipiche funzioni matematiche, come la radice quadrata o il logaritmo. Tra questi metodi utilizziamo il metodo `random()` che restituisce un numero generato in maniera casuale di tipo `double`, compreso tra 0 ed 0,9999999... (ovvero il numero `double` immediatamente più piccolo di 1). Nell'esercizio abbiamo moltiplicato per 4 questo numero, ottenendo un numero `double` casuale compreso tra 0 e 3,9999999... A questo viene applicato un cast ad intero e quindi vengono troncate tutte le cifre decimali. Abbiamo quindi ottenuto che la variabile `numeroVincente` immagazzini al runtime un numero generato casualmente, compreso tra 0 e 3, ovvero i possibili indici dell'array `griglia`.

Il metodo `corriGara()` genera quindi un numero casuale tra 0 e 3. Lo utilizza per individuare l'oggetto `Auto` dell'array `griglia` che vince la gara, per poi impostare il risultato tramite il metodo `dammiDettagli()` dell'oggetto `Auto` (che scriverà il lettore).

Tutti gli altri metodi della classe sono di tipo accessor e mutator.

Esercizio 5.c) Modificatori, static, e package, Vero o Falso:

1. Una classe dichiarata `private` non può essere utilizzata fuori dal package in cui è dichiarata.
2. La seguente dichiarazione di classe è scorretta:

```
public static class Classe { . . . }
```

3. La seguente dichiarazione di classe è scorretta:

```
protected class Classe { . . . }
```

4. La seguente dichiarazione di metodo è scorretta:

```
public void static metodo (){ . . . }
```

5. Un metodo statico può utilizzare solo variabili statiche e, perché sia utilizzato, non bisogna per forza istanziare un oggetto dalla classe in cui è definito.

6. Se un metodo è dichiarato `static`, non può essere chiamato al di fuori un package.
7. Una classe `static` non è accessibile fuori dal package in cui è dichiarata.
8. Un metodo `protected` viene ereditato in ogni sottoclasse qualsiasi sia il suo package.
9. Una variabile `static` viene condivisa da tutte le istanze della classe a cui appartiene.
10. Se non anteponiamo modificatori ad un metodo il metodo è accessibile solo all'interno dello stesso package.

Esercizio 5.d) Object Orientation in Java (pratica), Vero o Falso:

1. Un metodo statico deve essere per forza pubblico.
2. L'implementazione dell'incapsulamento implica l'utilizzo delle parole chiave `set` e `get`.
3. Per utilizzare le variabili incapsulate di una superclasse in una sottoclasse bisogna dichiararle almeno `protected`.
4. I metodi dichiarati privati non vengono ereditati nelle sottoclassi.
5. Un inizializzatore d'istanza viene invocato prima di ogni costruttore.
6. Una variabile privata risulta direttamente disponibile (teoricamente come se fosse pubblica) tramite l'operatore dot, a tutte le istanze della classe in cui è dichiarata.
7. La parola chiave `this` permette di referenziare i membri di un oggetto che sarà creato solo al runtime all'interno dell'oggetto stesso.
8. Se compiliamo la seguente classe:

```
public class CompilatorePensaciTu {  
    private int var;  
    public void setVar(int v) {  
        var = v;  
    }  
    public int getVar()  
    return var;  
}
```

il compilatore in realtà la trasformerà in:

```
import java.lang.*;  
  
public class CompilatorePensaciTu extends Object {  
    private int var;  
    public CompilatorePensaciTu() {  
    }  
    public void setVar(int v) {
```

```
this.var = v;  
}  
public int getVar()  
return this.var;  
}  
}
```

9. Compilando le seguenti classi, non si otterranno errori in compilazione:

```
public class Persona {  
private String nome;  
public void setNome(String nome) {  
this.nome = nome;  
}  
public String getNome() {  
return this.nome;  
}  
}  
  
public class Impiegato extends Persona {  
private int matricola;  
public void setMatricola(int matricola) {  
this.matricola = matricola;  
}  
public int getMatricola () {  
return this.matricola;  
}  
public String getDati() {  
return getNome() + "\nnumero" + getMatricola();  
}  
}
```

10. Alla classe `Impiegato` descritta nel punto 9) non è possibile aggiungere il seguente metodo:

```
public void setDati(String nome, int matricola) {  
setNome(nome);  
setMatricola(matricola);  
}
```

perché produrrebbe un errore in compilazione.

Soluzioni esercizi modulo 5

Esercizio 5.a) Object Orientation in generale (teoria), Vero o Falso:

1. **Falso**, esiste dagli anni '60.

2. **Vero**.

3. Falso. ogni linguaggio da supporto ai vari paradigmi in maniera diversa.

4. Vero.

5. Vero.

6. Vero.

7. Vero.

8. Vero.

9. Vero.

10. Falso, bisogna conoscere l'interfaccia pubblica e non l'implementazione interna.

Esercizio 5.b)

```
public class Pilota {  
    private String nome;  
  
    public Pilota(String nome) {  
        setNome(nome);  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
    public String getNome() {  
        return nome;  
    }  
}  
  
public class Auto {  
    private String scuderia;  
    private Pilota pilota;  
  
    public Auto (String scuderia, Pilota pilota) {  
        setScuderia(scuderia);  
        setPilota(pilota);  
    }  
    public void setScuderia(String scuderia) {  
        this.scuderia = scuderia;  
    }  
    public String getScuderia() {  
        return scuderia;  
    }  
    public void setPilota(Pilota pilota) {  
        this.pilota = pilota;  
    }  
    public Pilota getPilota() {  
        return pilota;  
    }  
}
```

```
public String dammiDettagli() {  
    return getPilota().getNome() + " su " + getScuderia();  
}  
}
```

Esercizio 5.c) Modificatori e package, Vero o Falso:

- 1. Falso**, `private` non si può utilizzare con la dichiarazione di una classe.
- 2. Vero**, `static` non si può utilizzare con la dichiarazione di una classe.
- 3. Vero**, `protected` non è applicabile a classi.
- 4. Vero**, `static` deve essere posizionato prima del `void`.
- 5. Vero**.
- 6. Falso**, `static` non è un modificatore d'accesso.
- 7. Falso**, `static` non è applicabile a classi.
- 8. Vero**.
- 9. Vero**.
- 10. Vero**.

Esercizio 5.d) Object Orientation in Java (pratica), Vero o Falso:

- 1. Falso**.
- 2. Falso**, non si tratta di parole chiave ma solo di parole utilizzate per convenzione.
- 3. Falso**, possono essere private ed essere utilizzate tramite i metodi accessor e mutator.
- 4. Vero**.
- 5. Vero**.
- 6. Vero**.
- 7. Vero**.
- 8. Vero**, anche se come vedremo più avanti, il costruttore di default non è vuoto.
- 9. Vero**.
- 10. Falso**.

**Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere le ragioni della nascita della programmazione ad oggetti (unità 5.1)	<input type="checkbox"/>	
Saper elencare i paradigmi ed i concetti fondamentali della programmazione ad oggetti (unità 5.2)	<input type="checkbox"/>	
Saper definire ed utilizzare il concetto di astrazione (unità 5.2)	<input type="checkbox"/>	
Comprendere l'utilizzo e l'utilità dell'incapsulamento (unità 5.3, 5.4)	<input type="checkbox"/>	
Comprendere l'utilizzo e l'utilità del reference <code>this</code> (unità 5.3)	<input type="checkbox"/>	
Capire quando e come usare l'incapsulamento (unità 5.4, 5.5)	<input type="checkbox"/>	
Comprendere l'utilizzo e l'utilità dei package (unità 5.6, 5.7)	<input type="checkbox"/>	
Capire e saper utilizzare i modificatori d'accesso (unità 5.8)	<input type="checkbox"/>	
Capire e saper utilizzare il modificatore <code>static</code> (unità 5.9)	<input type="checkbox"/>	

Note:

Ereditarietà e interfacce

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Comprendere l'utilizzo e l'utilità dell'ereditarietà (generalizzazione e specializzazione) (unità 6.1, 6.5).
- ✓ Conoscere le conseguenze dell'utilizzo contemporaneo di encapsulamento ed ereditarietà (unità 6.4).
- ✓ Comprendere e saper utilizzare i modificatori abstract e final (unità 6.3, 6.6).
- ✓ Comprendere e saper utilizzare la parola chiave super (unità 6.5).
- ✓ Comprendere e saper utilizzare il nuovo concetto di interfaccia in Java 8 (unità 6.7).
- ✓ Sapersi destreggiare nei vari casi complessi dell'ereditarietà multipla (unità 6.7).

In questo modulo affronteremo l'ereditarietà ed altri concetti ad essa correlati. Qui non parleremo solo ed esclusivamente di programmazione ma anche di concetti più astratti, dovremo avere una visione da progettisti. Chiunque può programmare, basta conoscere come si usa l'if e il for, e si può fare tutto! In teoria è così, in pratica bisogna avere delle regole e le idee chiare prima di "programmare a testa bassa". Quello che fa la differenza tra un bravo "smanettone" e un bravo sviluppatore è la qualità del suo lavoro. Uno sviluppatore dovrebbe saper analizzare e progettare la sua soluzione, creando codice semplice e manutenibile. Questo non è un libro sulla progettazione, ma cercheremo di stuzzicare la curiosità del lettore sull'argomento e cercare di farne comprendere l'importanza.

6.1 Ereditarietà

Come tutti i paradigmi che caratterizzano l'Object Orientation, anche il concetto di ereditarietà è ispirato a qualcosa che esiste nella realtà. Il termine è inteso in senso darwiniano. Nel mondo reale noi classifichiamo tutto con classi e sottoclassi. Per esempio un cane è un animale, un aereo è un veicolo, la chitarra è uno strumento musicale. In Java l'ereditarietà è la caratteristica che mette in relazione di estensibilità più classi che hanno caratteristiche comuni. Per esempio la classe `Cane` estenderà la classe `Animale`. Il risultato immediato è la possibilità di ereditare codice già scritto, e quindi gestire insiemi di classi collettivamente, giacché accomunate da alcune caratteristiche. Le regole per utilizzare correttamente l'ereditarietà sono semplici e chiare. Ma, sebbene l'ereditarietà sia un argomento agevole da comprendere, non è sempre utilizzata in maniera corretta.

6.1.1 La parola chiave extends

Consideriamo le seguenti classi che non incapsuliamo per semplicità:

```
public class Libro {  
    public String titolo;  
    public String autore;  
    public String editore;  
    public int numeroPagine;  
    public int prezzo;  
    . . .  
}  
  
public class LibroSuJava {  
    public String titolo;  
    public String autore;  
    public String editore;  
    public int numeroPagine;  
    public int prezzo;  
    public final String ARGOMENTO_TRATTATO = "Java";  
    . . .  
}
```

Notiamo che le classi `Libro` e `LibroSuJava` rappresentano due concetti in relazione tra loro e quindi dichiarano campi in comune. L'ereditarietà permetterà di mettere in relazione di estensione le due classi con la seguente sintassi:

```
public class LibroSuJava extends Libro {  
    public final String ARGOMENTO_TRATTATO = "Java";  
    . . .  
}
```

In questo modo la classe `LibroSuJava` erediterà tutti i campi pubblici della classe che estende. Quindi nella classe `LibroSuJava` sono presenti anche le variabili pubbliche `numeroPagine`, `prezzo`, `titolo`, `autore` ed `editore` definite nella classe `Libro`, anche se non sono state codificate esplicitamente. La costante `ARGOMENTO_TRATTATO` invece, appartiene solo alla classe `LibroSuJava`. In particolare diremo che `LibroSuJava` è **sottoclasse** di `Libro`, e `Libro` è **superclasse** di `LibroSuJava`. Il primo vantaggio è evidente, in questo esempio abbiamo riusato del codice evitando un copia-incolla, ma l'ereditarietà non aiuta solo a scrivere meno righe.

6.2 Il modificatore `final`

Abbiamo già visto come questo modificatore sia usato per dichiarare le costanti in Java. Ma `final` può essere utilizzato non solo con le variabili, ma anche con classi e metodi per influire sull'ereditarietà. Potremmo tradurre il termine `final` proprio con “finale”, nel senso di “non modificabile”. Infatti:

- ❑ una variabile dichiarata `final` diviene una costante;

- ❑ un metodo dichiarato `final` non può essere riscritto in una sottoclasse (non è possibile applicare l'override). Questo significa che si eredita così come è stato dichiarato nella superclasse;
- ❑ una classe dichiarata `final` non può essere estesa.
- ❑ Per esempio le classi `String` e la classe `Math` del package `java.lang` sono esempi di classi dichiarate `final`.

Infine il modificatore `final` si può utilizzare anche per variabili locali e parametri locali di metodi. In tali casi, i valori di tali variabili non saranno modificabili localmente. Per esempio, il seguente codice viene compilato senza errori:

```
public class LocalVariables {  
    public static void main(String args[]) {  
        System.out.println(new LocalVariables().  
            finalLocalVariablesMethod(5,6));  
    }  
    public int finalLocalVariablesMethod(final int i,final int j) {  
        final int k = i + j;  
        return k;  
    }  
}
```

6.3 La classe `Object`

Come abbiamo già osservato più volte, il compilatore Java inserisce spesso nel bytecode compilato alcune istruzioni che il programmatore non ha inserito, sempre al fine di agevolare lo sviluppatore sia nell'apprendimento sia nella codifica.

Abbiamo inoltre già asserito che la programmazione ad oggetti si ispira a concetti reali. Tutta la libreria standard di Java è stata pensata ed organizzata in maniera tale da soddisfare la teoria degli oggetti. Siccome la realtà è composta da oggetti (tutto può considerarsi un oggetto, sia elementi concretamente esistenti sia concetti astratti), nella libreria standard di Java esiste una classe chiamata `Object` che astrae il concetto di *oggetto generico*. Esso appartiene al package `java.lang` ed è di fatto la superclasse di ogni classe. È in cima alla gerarchia delle classi e quindi tutte le classi ereditano i membri di `Object` (è possibile verificare quest'affermazione dando uno sguardo alla documentazione). Se definiamo una classe che non estende altre classi, essa automaticamente estenderà `Object`. Ciò significa che se scrivessimo:

```
public class Arte {  
    . . .  
}
```

il compilatore in realtà interpreterebbe questa classe come fosse scritta nel seguente modo:

```
public class Arte extends Object {  
    . . .
```

Nel mondo reale tutto è un oggetto, quindi in Java tutte le classi estenderanno `Object`.

Quindi quando creiamo una nostra classe, questa viene inserita nella gerarchia di classe alla cui cima c'è `Object`. Ogni classe quindi eredita tutti i metodi definiti dalla classe `Object`, ma di questo parleremo nei prossimi moduli.

6.4 Rapporto ereditarietà-incapsulamento

Dal momento che l'incapsulamento si può considerare obbligatorio e l'ereditarietà un prezioso strumento di sviluppo, bisognerà chiedersi che cosa provocherà l'utilizzo combinato di entrambi i paradigmi. Ovvvero: che cosa erediteremo da una classe encapsulata? Abbiamo già affermato alla fine del modulo precedente che estendere una classe significa ereditarne i membri non privati. Consideriamo quindi una classe `Ricorrenza` ottenuta specializzando la classe `Data` definita nel modulo precedente come esempio di encapsulamento. È escluso che la classe `Ricorrenza` possa accedere alle variabili `giorno`, `mese` e `anno` direttamente, giacché queste non saranno ereditate. Infatti tali variabili erano state dichiarate private all'interno della superclasse `Data`. Ma essendo tutti i metodi "set" e "get" dichiarati pubblici nella superclasse, verranno ereditati e quindi utilizzabili nella sottoclass. Concludendo, anche se la classe `Ricorrenza` non possiede esplicitamente le variabili private di `Data`, può comunque usufruirne tramite l'incapsulamento. In pratica è come se le possedesse.

Quindi non è necessario dichiarare una variabile `protected` per ereditarla nelle sottoclassi. Avere a disposizione i metodi mutator (set) ed accessor (get) nelle sottoclassi è più che sufficiente. Inoltre dichiarare protetta una variabile d'istanza significa renderla pubblica a tutte le classi dello stesso package (cfr. paragrafo 5.8). Questo significa che la variabile sarà accessibile direttamente per le classi appartenenti allo stesso package, e quindi non sarà veramente encapsulata.

6.5 Quando utilizzare l'ereditarietà

Quando si parla di ereditarietà si è spesso convinti che per implementarla basti avere un paio di classi che dichiarino campi in comune. In realtà ciò potrebbe essere interpretato come "indizio" di ereditarietà. Il test decisivo deve però essere effettuato mediante la cosiddetta "is a" relationship (la relazione "è un").

6.5.1 La relazione "is a"

Per un corretto uso dell'ereditarietà, il programmatore deve porsi solo una semplice e fondamentale domanda: un oggetto della candidata sottoclasse "è un" oggetto della candidata superclasse? Se la risposta alla domanda è negativa, l'ereditarietà non si deve utilizzare. Effettivamente, se l'applicazione dell'ereditarietà dipendesse solamente dai campi in comune tra due classi, potremmo trovare relazioni d'estensione tra classi quali `Triangolo` e `Rettangolo`. Per esempio:

```

public class Triangolo {
    public final int NUMERO_LATI = 3;
    public float lunghezzaLatoUno;
    public float lunghezzaLatoDue;
    public float lunghezzaLatoTre;
    //...
}

public class Rettangolo extends Triangolo {
    public final int NUMERO_LATI = 4;
    public float lunghezzaLatoQuattro;
    //...
}

```

ma un rettangolo non è un triangolo, e per la relazione “is a” quest'estensione non è valida. Il problema è che la costruzione di un programma avviene passo dopo passo. Se mettiamo in relazione di ereditarietà in maniera errata due classi, in un primo momento risparmieremmo del codice, ma i problemi sorgerebbero dopo, quando tenteremo di sfruttare un rettangolo così come sfruttiamo un triangolo. La relazione “è un” ci mette al sicuro da questi inconvenienti ed è molto semplice da testare.

Si noti che i problemi logici di un'astrazione scorretta dei dati potrebbero essere propagati al codice implementando ereditarietà e encapsulamento. On line trovate l'appendice G con un esempio guidato alla programmazione ad oggetti, che lo dimostrerà.

L'ereditarietà è quindi definita dalla relazione “è un”, e quest'ultima si applica agli oggetti non alle classi.

6.5.2 Ereditarietà e costruttori

L'ereditarietà non è applicabile ai costruttori. Anche quando sono dichiarati pubblici, i costruttori non sono ereditati per un motivo molto semplice: il loro nome. Ricordiamo che un costruttore è stato definito nel modulo 2 come metodo speciale, in quanto possiede le seguenti proprietà:

1. ha lo stesso nome della classe cui appartiene;
2. non ha tipo di ritorno;
3. è chiamato automaticamente (e solamente) ogni volta che viene istanziato un oggetto della classe cui appartiene, relativamente a quell'oggetto;
4. è presente in ogni classe.

Relativamente all'ultimo punto abbiamo anche definito il “costruttore di default” come il costruttore che è introdotto in una classe dal compilatore al momento della compilazione, nel caso il programmatore non gliene abbia fornito uno in maniera esplicita. Abbiamo anche affermato che

soltamente un costruttore è utilizzato per inizializzare le variabili degli oggetti al momento dell'istanza. Detto questo, se per esempio la classe `LibroSuJava` ereditasse dalla classe `Libro` un costruttore, erediterebbe un costruttore che si chiama proprio `Libro()`. Un costruttore che si chiama `Libro()`, in una classe che si chiama `LibroSuJava` non potrà mai essere chiamato! Infatti per istanziare un oggetto dalla classe `LibroSuJava` è necessario obbligatoriamente chiamare un costruttore chiamato `LibroSuJava()`. Per esempio, consideriamo la seguente istruzione:

```
LibroSuJava libroSuJava = new LibroSuJava();
```

Se al suo posto scrivessimo:

```
LibroSuJava libroSuJava = new Libro();
```

istanzieremmo un oggetto della classe `Libro` e non un oggetto della classe `LibroSuJava`.

Il fatto che i costruttori non siano ereditati dalle sottoclassi è assolutamente in linea con la sintassi del linguaggio, ma contemporaneamente è in contraddizione con i principi della programmazione ad oggetti. In particolare sembra violata la regola dell'astrazione. Infatti, nel momento in cui lo sviluppatore ha deciso di implementare il meccanismo dell'ereditarietà, ha dovuto testarne la validità mediante la cosiddetta relazione “*is a*”. Alla domanda: “un oggetto istanziato dalla candidata sottoclasse può considerarsi anche un oggetto della candidata superclasse?” ha infatti risposto affermativamente.

Un libro su Java, essendo anche un libro, deve avere tutte le caratteristiche di un libro. In particolare deve riutilizzarne anche il costruttore. Non potendolo ereditare però, l'astrazione sembra violata. Invece è proprio in una situazione del genere che Java dimostra la sua coerenza. Aggiungiamo infatti un'altra proprietà alla definizione di costruttore:

un qualsiasi costruttore (anche quello di default), come prima istruzione, invoca sempre un costruttore della superclasse.

Per esempio, aggiungiamo dei costruttori alle classi `Libro` e `LibroSuJava`:

```
public class Libro {  
    ...  
    public Libro (){  
        System.out.println("Costruito un Libro!");  
    }  
}  
  
public class LibroSuJava extends Libro {  
    public LibroSuJava (){  
        System.out.println("Costruito un Libro su Java!");  
    }  
}
```

Il lettore, avendo appreso che i costruttori non sono ereditati, dovrebbe concludere che l'istanza di un `LibroSuJava`, mediante una sintassi del tipo:

```
new LibroSuJava(); /* N.B.: L'assegnazione di un reference non è richiesta  
per istanziare un oggetto */
```

produrrebbe in output la seguente stringa:

```
Costruito un Libro su Java!
```

L'output risultante sarà invece:

```
Costruito un Libro!  
Costruito un Libro su Java!
```

Infatti il costruttore `LibroSuJava()` ha prima invocato il costruttore della superclasse `Libro()` e poi è stato eseguito.

La chiamata obbligatoria ad un costruttore di una superclasse viene effettuata tramite la parola chiave `super`, che viene introdotta di seguito.

6.5.3 La parola chiave `super`

Nel modulo 5 abbiamo definito la parola chiave `this` come *reference implicito all'oggetto corrente*. Possiamo definire la parola chiave `super` come *reference implicito all'intersezione tra l'oggetto corrente e la sua superclasse*. In pratica questo reference ci permette di accedere ai componenti della superclasse ed in particolare al costruttore.

La parola chiave `super` è strettamente legata al concetto di costruttore. In ogni costruttore, infatti, è **sempre** presente una chiamata al costruttore della superclasse tramite una sintassi speciale che sfrutta il reference `super`. Per esempio nella classe `LibroSuJava` il costruttore verrà modificato dal compilatore nel seguente modo:

```
public class LibroSuJava extends Libro {  
    public LibroSuJava () {  
        super(); //istruzione隐式的 se non fornita esplicitamente.  
        System.out.println("Costruito un Libro su Java!");  
    }  
}
```

Ecco come il costruttore della classe `Libro` viene invocato dal costruttore della classe `LibroSuJava`. Possiamo esplicitare la chiamata a `super()`, ma se non lo facciamo il compilatore considererà questa istruzione implicita. La chiamata ad un costruttore della superclasse è quindi inevitabile!

Si noti che anche il costruttore della classe `Libro` chiamerà il costruttore della sua superclasse `Object` mediante un comando `super()` implicito.

Supponiamo ora di voler dotare la superclasse `Libro` di un costruttore che imposti la variabile `titolo`. D'altronde un oggetto di tipo `Libro` almeno un titolo lo deve avere:

```
public class Libro {  
    . . .  
    public Libro (String titolo) {  
        this.titolo = titolo;  
    }  
}
```

Questa classe compilerà, ma `LibroSuJava` non compilerà più:

```
D:\java8\Codice\modulo_06\esempi\LibroSuJava.java:1:  
error: constructor Libro in class Libro cannot be applied to given types;  
public class LibroSuJava extends Libro{  
^  
    required: String  
    found: no arguments  
reason: actual and formal argument lists differ in length  
1 error
```

Modifichiamo la superclasse e non compila la sottoclasse, strano! A pensarci bene non è nemmeno tanto strano visto che sono legate dalla relazione “is a”. Leggendo l'errore del compilatore comunque si può intuire quello che è successo. Il costruttore della sottoclasse ha cercato di chiamare un costruttore della superclasse che non c'è: il costruttore senza parametri. Infatti, se un costruttore viene aggiunto esplicitamente, nessun altro costruttore di default viene implicitamente inserito. Quindi l'istruzione `super()` inserita implicitamente nella sottoclasse `LibroSuJava` prova a chiamare il costruttore della superclasse senza parametri (infatti non sono passati parametri tra le parentesi). Per risolvere il problema bisogna modificare il costruttore di `LibroSuJava` in modo tale da fargli chiamare il costruttore della superclasse `Libro` che prende in input il titolo. La soluzione seguente è la migliore:

```
public class LibroSuJava extends Libro {  
    public LibroSuJava (String titolo) {  
        super(titolo);  
    }  
    // . . .  
}
```

Abbiamo quindi inserito una chiamata esplicita al costruttore della superclasse che prende in input una stringa. Ci è venuto naturale modificare anche la lista dei parametri della sottoclasse, ed anche questo può essere considerato indice della coerenza di Java verso i paradigmi object oriented.

Attenzione: la chiamata al costruttore della superclasse mediante `super()` deve essere la prima istruzione di un costruttore e non potrà essere inserita all'interno di un metodo che non sia un costruttore.

Se avessimo più costruttori nella superclasse potremmo scegliere quale chiamare. Per esempio se la classe `Libro` definisse più costruttori nel seguente modo:

```
public class Libro {  
    // ...  
    public Libro (String titolo, String autore) {  
        this(titolo); // Chiamata al secondo costruttore  
        setAutore(autore);  
    }  
  
    public Libro (String titolo) {  
        this.titolo = titolo;  
    }  
    // ...  
}
```

La classe `LibroSuJava` potrebbe chiamare a seconda del caso il costruttore più adeguato. Per esempio:

```
public class LibroSuJava extends Libro {  
    public LibroSuJava (String titolo) {  
        super(titolo);  
    }  
  
    public LibroSuJava (String titolo, String autore) {  
        super(titolo, autore);  
    }  
  
    // ...  
}
```

La chiamata tramite `super()` ad un costruttore di una superclasse o tramite `this()` ad un costruttore della classe stessa, può essere usata solo come prima istruzione in un costruttore. Se una di queste istruzioni non fosse inserita come prima istruzione, il compilatore segnalerebbe un errore. Questo significa che non è neanche possibile chiamare un costruttore tramite questi comandi da un metodo che non sia un costruttore. Inoltre se si chiama esplicitamente un altro costruttore della stessa classe mediante il comando `this()`, allora la chiamata a `super()` non sarà inserita nello stesso costruttore chiamante. Si noti però che la chiamata ad un altro costruttore della stessa classe mediante `this()` rimanda solamente la chiamata al costruttore della superclasse. Infatti questa è comunque inserita esplicitamente o implicitamente all'interno del costruttore chiamato (come prima istruzione).

Visto che `super` e `this` possono addirittura chiamare costruttori, possibile che non possano chiamare metodi? Possono farlo e per il `this` lo abbiamo già visto nel precedente modulo. Con la sitassi:

```
this.nomeMetodo()
```

era possibile chiamare metodi della stessa classe, ma avevamo anche detto che il reference `this` in questi casi è ridondante visto che comunque Java lo considera implicito se non è specificato un reference prima della chiamata di un metodo. Con la sintassi:

```
super.nomeMetodo()
```

sarà possibile invocare da un metodo di una sottoclasse un metodo di una superclasse. Questo ha senso in caso di riscrittura di un metodo in una sottoclasse, altrimenti il metodo della superclasse sarà ereditato anche nella sottoclasse, e quindi non avrebbe molto senso. Per esempio consideriamo le seguenti classi:

```
public class Persona {  
    private String nome, cognome;  
    public String toString(){  
        return nome + " " + cognome;  
    }  
    . . .  
    //accessor e mutator methods (set e get)  
}  
  
public class Cliente extends Persona {  
    private String indirizzo, telefono;  
    public String toString() {  
        return super.toString() + "\n" +  
            indirizzo + "\nTel:" + telefono;  
    }  
    . . .  
    //accessor e mutator methods (set e get)  
}
```

La classe `Persona` dichiara un metodo che si chiama `toString()`. Si tratta di uno dei metodi più famosi in quanto dichiarato direttamente dalla classe `Object`, e dovrebbe essere il metodo responsabile a rappresentare con una stringa l'oggetto su cui viene chiamato il metodo. Quindi la classe `Persona` sta già facendo override (riscrittura) del metodo, facendo sì che il metodo ritorni una stringa composta dal `nome` e dal `cognome`.

Se il metodo `toString()` non fosse stato riscritto, avrebbe ritornato una stringa con il formato `NomeClasse@IndirizzoInFormatoEsadecimale`. Questo perché la classe `Object` è troppo generica, e quindi i progettisti di Java decisero di restituire una stringa descrittiva del reference.

La classe `Cliente` estende la classe `Persona` e ne eredita anche il metodo `toString()`. Ma questo metodo viene a sua volta ridefinito per stampare anche i nuovi campi definiti nella classe `cliente`:

indirizzo e telefono. Per stampare anche il nome e il cognome, il metodo `toString()` della classe `Cliente` in primo luogo chiama il metodo `toString()` della superclasse `Persona`, e poi l'aggancia con il resto della stringa. Ricordiamo che la stringa “`\n`” equivale ad andare a capo come abbiamo visto nel modulo 3.

L'override sarà ampiamente approfondito nel prossimo modulo.

6.5.4 Ereditarietà e inizializzatori

Anche gli inizializzatori statici e d'istanza sembra che vengano in qualche modo ereditati. In realtà non si tratta di ereditarietà vera e propria, ma semmai di normale amministrazione. Dimostriamolo direttamente con un esempio. Consideriamo una classe che definisce due semplici inizializzatori (uno statico e uno d'istanza) oltre ad un costruttore senza parametri:

```
public class ParentInitializer {  
    public ParentInitializer () {  
        System.out.println("Costruttore di ParentInitializer");  
    }  
    {  
        System.out.println("Inizializzatore d'istanza di " + "ParentInizializer");  
    }  
  
    static {  
        System.out.println("Inizializzatore statico di " + "ParentInizializer");  
    }  
}
```

se estendiamo `ParentInitializer` con la seguente classe che dichiara a sua volta altri due inizializzatori e con un metodo `main()` si auto-istanzia:

```
public class ChildInitializer extends ParentInitializer {  
    public ChildInitializer () {  
        System.out.println("Costruttore di ChildInitializer");  
    }  
    {  
        System.out.println("Inizializzatore d'istanza di " + "ChildInizializer");  
    }  
  
    static {  
        System.out.println("Inizializzatore statico di " + "ChildInizializer");  
    }  
    public static void main(String args[]) {  
        new ChildInitializer();  
    }  
}
```

l'output risultante sarà il seguente:

```
Inizializzatore statico di ParentInizializer  
Inizializzatore statico di ChildInizializer  
Inizializzatore d'istanza di ParentInizializer  
Costruttore di ParentInizializer  
Inizializzatore d'istanza di ChildInizializer  
Costruttore di ChildInizializer
```

Si noti che per gli inizializzatori sia statici che d'istanza non valgono i modificatori d'accesso, e quindi non valgono le regole dell'ereditarietà. Quello che accade è che prima che sia caricata in memoria la classe `ChildInitializer`, deve essere caricata in memoria la classe `ParentInitializer`, e questo spiega le prime due righe dell'output. Ricordiamo che mentre gli inizializzatori statici vengono invocati al momento del caricamento in memoria della classe, gli inizializzatori d'istanza vengono chiamati al momento dell'istanza dell'oggetto, prima del costruttore. Nell'esempio capita che sia chiamato il costruttore di `ChildInitializer` che come abbiamo visto chiama il costruttore di `ParentInitializer`. Ma prima che questo sia eseguito, viene eseguito l'inizializzatore di `ParentInitializer`. Lo stesso accade nella sottoclasse, dove prima che sia chiamato il costruttore viene invocato l'inizializzatore, ed ecco spiegato l'output del precedente esempio.

In realtà gli inizializzatori si usano raramente ma è bene sapere cosa succede in caso di ereditarietà.

6.5.5 Generalizzazione e specializzazione

Sono due termini che definiscono i processi che portano all'implementazione dell'ereditarietà.

Si parla di generalizzazione se, a partire da un certo numero di classi, si definisce una superclasse che ne raccoglie le caratteristiche comuni.

Viceversa si parla di specializzazione quando partendo da una classe, si definiscono una o più sottoclassi allo scopo di ottenere oggetti più specializzati.

L'utilità della specializzazione è evidente: supponiamo di voler creare una classe `MioBottone`, le cui istanze siano visualizzate come pulsanti da utilizzare su un'interfaccia grafica. Partire da zero creando pixel per pixel è molto complicato. Se invece estendiamo la classe `Button` del package `java.awt` dobbiamo solo aggiungere il codice che personalizzerà il `MioBottone`.

Nell'ultimo esempio del paragrafo 6.5.1 avevamo a disposizione la classe `Triangolo` e la classe `Rettangolo`. Abbiamo notato come il test "is a", fallendo, ci sconsigli l'implementazione dell'ereditarietà. Eppure queste due classi hanno campi in comune e non sembra che sia un evento casuale. In effetti sia il `Triangolo` sia il `Rettangolo` sono ("is a") entrambi poligoni.

La soluzione a questo problema è *naturale*. Basta generalizzare le due astrazioni in una classe `Polygon`, che potrebbe essere estesa dalle classi `Triangolo` e `Rettangolo`. Per esempio:

```
public class Poligono {  
    public int numeroLatini;  
    public float lunghezzaLatoUno;
```

```
public float lunghezzaLatoDue;
public float lunghezzaLatoTre;
. . .
}

public class Triangolo extends Poligono {
public final int NUMERO_LATI = 3;
. . .
}

public class Rettangolo extends Poligono {
public final int NUMERO_LATI = 4;
public float lunghezzaLatoQuattro;
. . .
}
```

Se fossimo partiti dalla classe `Poligono` per poi definire le due sottoclassi, avremmo parlato invece di specializzazione. Usando il processo di specializzazione è facile pensare anche ad altre eventuali sottoclassi come `Esagono`, `Pentagono`, `Ottagono`, `Quadrato` e così via. La classe `Poligono` quindi è la più generica, e in un eventuale programma che usi poligoni, l'unica classe che probabilmente non istanzieremo sarà proprio la classe `Poligono`. In questi casi la classe `Poligono` va dichiarata `abstract` e questo implicherà che non potrà essere istanziata (avremmo un errore in compilazione se ci provassimo). Quindi `Poligono` rimarrebbe importante solo per permetterci di sfruttare i vantaggi dell'ereditarietà ma non esisteranno oggetti istanziati direttamente da questa classe. Tuttavia, per quanto detto, oggetti istanziati dalle sue sottoclassi (`Quadrato`, `Triangolo` etc.) potranno essere considerati tutti poligoni!

6.6 Il modificatore `abstract`

Il modificatore `abstract` può essere applicato non solo a classi ma anche a metodi.

6.6.1 Metodi astratti

Un metodo astratto non implementa un proprio blocco di codice, e quindi è privo di comportamento. Nella pratica la definizione di un metodo astratto non definisce parentesi graffe, ma termina con un punto e virgola. Un esempio di metodo astratto potrebbe essere il seguente:

```
public abstract void dipingiQuadro();
```

Questo metodo non potrà essere invocato (in quanto non è definito) ma potrà essere soggetto a riscrittura (`override`) in una sottoclasse come vedremo approfonditamente nel prossimo modulo dedicato al polimorfismo.

Inoltre un metodo astratto potrà essere definito solamente all'interno di una classe astratta. In altre parole una classe che contiene anche un solo metodo astratto deve essere dichiarata astratta.

6.6.2 Classi astratte

Una classe dichiarata astratta non può essere istanziata. Il programmatore che ha intenzione di marcare una classe con il modificatore `abstract` deve essere consapevole a priori che da quella classe non saranno istanziabili oggetti. Consideriamo la seguente classe astratta:

```
public abstract class Pittore {  
    //...  
    public abstract void dipingiQuadro();  
    //...  
}
```

Questa classe ha senso se inserita in un sistema in cui l'oggetto `Pittore` può essere considerato troppo generico per definire un nuovo tipo di dato da istanziare. Supponiamo che per il nostro sistema sia fondamentale conoscere lo stile pittorico di un oggetto `Pittore` e, siccome non esistono pittori capaci di dipingere con un qualsiasi tipo di stile tra tutti quelli esistenti, non ha senso istanziare una classe `Pittore`. Sarebbe corretto popolare il nostro sistema di sottoclassi non astratte di `Pittore` come `PittoreImpressionista` e `PittoreNeoRealista`. Queste sottoclassi devono ridefinire il metodo astratto `dipingiQuadro` (a meno che non si abbia l'intenzione di dichiarare astratte anche esse). Si noti che la classe `Pittore` avrebbe potuto implementare concretamente il metodo `dipingiQuadro`, ma a livello logico non sarebbe giusto definirlo favorendo uno stile piuttosto che un altro. Nulla vieta però ad una classe astratta di implementare tutti suoi metodi.

Facciamo un esempio parlando di strumenti musicali:

```
public abstract class Strumento { //Classe astratta  
    public String nome;  
    public String prezzo;  
    public abstract void suonaFaDiesis(); /*Ogni strumento suona in modo  
        diverso! Impossibile definire questo metodo!*/  
    //...  
}  
  
public class Chitarra extends Strumento { // Classe concreta  
    public void suonaFaDiesis() { // Override (riscrittura) del metodo  
        //Implementazione del metodo per la chitarra.  
    }  
    //...  
}  
  
public abstract class StrumentoAFiato extends Strumento { //Classe di  
//nuovo astratta che estende Strumento  
//metodo suonaFaDiesis ereditato ancora astratto e  
//non riscritto perché troppo generico!  
    //...  
}  
  
public class Flauto extends StrumentoAFiato { // Classe concreta che  
//estende StrumentoAFiato  
    public void suonaFaDiesis() {  
        //Implementazione del metodo per il flauto.  
    }
```

```
}
```

```
// . . .
```

```
}
```

In questo esempio abbiamo visto come `Strumento` sia la superclasse della classe concreta `Chitarra` (che ridefinisce il metodo `suonaFaDiesis()`) e della classe astratta `StrumentoAFiato` (che non ridefinisce il metodo `suonaFaDiesis()` che viene ereditato astratto). Infine la classe concreta `Flauto` estende `StrumentoAFiato` (che a sua volta estendeva `Strumento`). Quindi la chitarra è uno strumento e il flauto è uno strumento a fiato, ma è anche uno strumento: è tutto coerente! Sapreste aggiungere a questa gerarchia una classe `StrumentoACorde` e una classe `Sassofono`?

Il grande vantaggio che offre l'implementazione di una classe astratta è che “obbliga” le sue sottoclassi ad implementare un comportamento. A livello di progettazione le classi astratte costituiscono uno strumento fondamentale.

Il modificatore `abstract`, per quanto riguarda le classi, potrebbe essere considerato l'opposto del modificatore `final`. Infatti, una classe `final` non può essere estesa, mentre una classe dichiarata `abstract` deve essere estesa. Non è possibile utilizzare congiuntamente per classi e metodi i modificatori `abstract` e `final`, per chiari motivi logici. Non è altrettanto possibile utilizzare per metodi congiuntamente i modificatori `abstract` e `static`.

A differenza del mondo reale, dove una persona potrebbe essere contemporaneamente un genitore, un programmatore, un lettore di libri e un musicista, in Java non esiste propriamente quella che viene definita “ereditarietà multipla”: una classe può estendere solo una classe per volta. Questo significa che una classe `Persona` non può essere dichiarata come nel seguente esempio:

```
public class Persona  
    extends Programmatore, Genitore, Lettore, Musicista { // Illegale!
```

Bisognerà scegliere solo una superclasse in quanto in Java dopo la clausola `extends` è possibile specificare solo una classe. Le ragioni per cui non è stata permessa l'ereditarietà multipla in Java risiedono nelle precedenti esperienze di altri linguaggi, i quali hanno dovuto sopportare con stratagemmi complessi ai limiti della definizione. C'è però una grossa novità in Java 8! Con il nuovo concetto di interfaccia, sarà possibile avere una ereditarietà multipla “soft”, che non introduce i problemi presenti in altri linguaggi. Intanto rimane valida la regola che è possibile estendere una sola classe per volta.

6.7 Interfacce

Dal punto di vista della progettazione un'interfaccia è un'evoluzione del concetto di classe astratta. Le convenzioni per il codice sono le stesse che valgono per le classi.

6.7.1 Definizione classica (pre-Java 8)

Sino a Java 7 un'interfaccia per definizione poteva possedere solo metodi dichiarati implicitamente `public` e `abstract`, e variabili dichiarate implicitamente `public`, `static` e `final`. Questo indipendentemente dal fatto che questi modificatori siano usati oppure no. Di seguito un primo semplice esempio:

```
public interface Saluto {  
    public static final String CIAO = "Ciao";  
    public static final String BUONGIORNO = "Buongiorno";  
    . . .  
    public abstract void saluta();  
}
```

che è equivalente a:

```
public interface Saluto {  
    String CIAO = "Ciao";  
    String BUONGIORNO = "Buongiorno";  
    . . .  
    void saluta();  
}
```

Anche in questo caso Java lavora per noi “dietro le quinte” aggiungendo i modificatori che non abbiamo scritto. Quindi che si voglia o no, `CIAO` e `BUONGIORNO` sono sempre costanti statiche pubbliche e il metodo `saluta()` è sempre `public` e `abstract` (d'altronde il metodo termina con un punto e virgola). Questo significa che non è possibile usare modificatori in contrasto con quelli che Java si attende. Per esempio se provassimo a dichiarare il metodo `saluta()` come `protected`, otterremmo un errore in compilazione.

```
D:\java8\Codice\modulo_06\esempi\Saluto.java:5: error:  
modifier protected not allowed here  
protected void saluta();  
^  
1 error
```

In qualsiasi caso ogni membro (variabile o metodo) di una classe astratta ha visibilità pubblica.

Come le classi, le interfacce si devono scrivere all'interno di file che hanno esattamente lo stesso nome dell'interfaccia che definiscono, con suffisso “.java”. Quindi l'interfaccia `Saluto` dell'esempio deve essere salvata all'interno di un file di nome “`Saluto.java`”.

Un'interfaccia non si può istanziare (non è una classe). Per essere utilizzata ha bisogno di essere in qualche modo estesa. Ma solo un'altra interfaccia può estendere un'altra interfaccia, e quindi è possibile creare gerarchie costituite da sole interfacce. Ma senza istanziare oggetti i programmi non funzionano, e quindi abbiamo bisogno che le classi ereditino dalle interfacce. Ecco che allora viene

in aiuto la parola chiave `implements`, che si usa in maniera simile ad `extends`. La differenza tra implementare un'interfaccia ed estendere una classe consiste essenzialmente nel fatto che, mentre possiamo estendere una sola classe alla volta, possiamo invece implementare un numero indefinito di interfacce simulando di fatto l'ereditarietà multipla (che verrà affrontata in dettaglio nei prossimi paragrafi).

Un'interfaccia, essendo un'evoluzione di una classe astratta, non può essere istanziata. Potremmo utilizzare l'interfaccia dell'esempio nel seguente modo:

```
public class SalutoImpl implements Saluto {  
    public void saluta() {  
        System.out.println(CIAO);  
    }  
}
```

Riassumendo, una classe può implementare un'interfaccia e un'interfaccia può estendere un'altra interfaccia.

Prima dell'avvento di Java 8, il concetto di interfaccia è stato sempre molto apprezzato per la sua semplicità. Se una classe ereditava metodi da un'interfaccia, allora doveva “onorare il contratto” e ridefinire i metodi astratti ereditati (a meno che non si volesse dichiarare astratta anche la classe). L'uso di un'interfaccia è sempre stato limitato alla sua funzione di essere una specifica astratta per le classi che la implementavano. Ma con il passare del tempo i linguaggi di programmazione hanno bisogno di evolversi per soddisfare i bisogni degli sviluppatori. Negli anni ogni evoluzione di Java ha portato sia benefici che costi. Oggi con Java 8 le interfacce hanno acquisito maggiore potenza, ma contemporaneamente le regole che ne governano l'utilizzo si sono complicate.

6.7.2 Metodi statici (Java 8)

Con Java 8 è possibile definire all'interno delle interfacce anche metodi statici. In effetti non c'è mai stato un problema tecnico per il quale non si è mai voluto inserire metodi statici all'interno delle interfacce. Semplicemente non sembrava in linea con la concezione di interfaccia come contratto da implementare. Quindi oggi è possibile scrivere interfacce nel seguente modo:

```
public interface StaticMethodInterface {  
    static void metodoStatico() {  
        System.out.println("Metodo Statico Chiamato!");  
    }  
}
```

e chiamare metodi statici direttamente dalle interfacce:

```
public class TestStaticMethodInterface {  
    public static void main(String args[]) {  
        StaticMethodInterface.metodoStatico();  
    }  
}
```

Usando le interfacce contenenti metodi statici ne tradiamo la natura contrattuale per la quale fu definita. Un'interfaccia come la precedente non ha bisogno di essere implementata, infatti i metodi statici di un'interfaccia non vengono ereditati. Il comando **implements** non fa ereditare i metodi statici. Interfacce di questo tipo servono semplicemente per definire metodi statici e pubblici, ovvero definiscono funzioni. Per esempio se considerassimo la seguente classe:

```
public class StaticMethodClass implements StaticMethodInterface {  
}
```

allora il seguente codice:

```
public class TestStaticMethodInterface {  
    public static void main(String args[]) {  
        StaticMethodClass.metodoStatico();  
    }  
}
```

produrrebbe un errore in compilazione:

```
D:\java8\Codice\modulo_06\esempi\TestStaticMethodInterface  
.java:3: error: cannot find symbol  
StaticMethodClass.metodoStatico();  
^  
symbol: method metodoStatico()  
location: class StaticMethodClass  
1 error
```

Invece è necessario chiamare il metodo sfruttando il nome dell'interfaccia e non il nome della sua sottoclasse:

```
public class TestStaticMethodInterface {  
    public static void main(String args[]) {  
        StaticMethodInterface.metodoStatico();  
    }  
}
```

Prima di Java 8 per ottenere strutture con metodi statici sono sempre state usate le classi, magari astratte o con costruttori privati (come la classe `Math` di cui abbiamo già accennato qualcosa) o implementate come Singleton (cfr. paragrafo 5.9.6). Usare un'interfaccia popolata di metodi statici in luogo di una classe non sembra essere un grosso vantaggio. Una classe potrebbe dichiarare per esempio un inizializzatore statico per eseguire del codice nel momento in cui è caricata in memoria per la prima volta. In un'interfaccia invece è illegale dichiarare inizializzatori.

6.7.3 Metodi di default e interfacce funzionali (Java 8)

Altra novità di Java 8 è la possibilità di dichiarare metodi concreti all'interno delle interfacce. Si parla di **metodi di default** (in inglese *default methods*) perché vengono dichiarate usando come modificatore la parola chiave `default`. Per esempio consideriamo il seguente codice:

```
public interface Solista {  
    default void eseguiAssolo() {  
        //Scala maggiore in DO  
        System.out.println("DO RE MI FA SOL LA SI");  
    }  
}
```

In questo modo possiamo ereditare questo metodo in un'eventuale sottoclasse senza dover riscrivere il metodo (che non è astratto). La seguente classe compila senza problemi:

```
public class Musicista implements Solista {  
}
```

Il metodo `eseguiAssolo()` sarà sempre possibile riscriverlo all'occorrenza (ma questo è un argomento che approfondiremo nel prossimo modulo). Avere però un'implementazione di default può essere in generale molto comodo.

È possibile scrivere più di un metodo di default in un'interfaccia, e questi possono convivere con metodi astratti e metodi statici. Il nome “interfaccia” in un certo senso ha perso di significato, anche se è sempre possibile usare le interfacce dichiarando solo metodi astratti. In particolare prendono il nome di **interfacce funzionali** (in inglese **functional interfaces**) le interfacce che contengono un unico metodo astratto. Per indicare quest'ultimo viene spesso usato l'acronimo **SAM**, che sta per **Single Abstract Method** (in italiano **singolo metodo astratto**). Vengono chiamate così proprio perché è come se esistessero solo per dichiarare una funzione da implementare. Esiste un nuovo package che dichiara una serie di interfacce funzionali: il package `java.lang.function`. Anche sulle interfacce funzionali torneremo più avanti per spiegare le loro applicazioni, in particolare nel Modulo 15 dedicato alle espressioni lambda.

6.7.4 Ereditarietà multipla

Come abbiamo già detto una classe può implementare più interfacce. Quindi in Java oggi esiste una manifestazione semplificata di quella caratteristica chiamata **ereditarietà multipla**.

Parliamo di semplificazione perché in realtà le classi possono ereditare dalle interfacce solo la loro parte funzionale (i metodi) e non i dati (a parte le costanti statiche che un'interfaccia può dichiarare). In altri linguaggi invece esiste l'ereditarietà multipla completa, ma anche regole molto complicate per gestire i problemi derivanti dalla sua implementazione.

Era possibile implementare più interfacce anche prima di Java 8, ma si ereditavano metodi astratti che bisognava riscrivere. Era tutto molto semplice anche nel caso si ereditassero due metodi con lo stesso nome da interfacce diverse: era sempre necessario riscrivere il metodo. Con l'avvento dei metodi di default l'ereditarietà multipla ha quindi assunto un significato diverso rispetto al passato. Quindi se consideriamo le seguenti interfacce:

```
public interface Lettore {  
    default void leggi(Libro libro) {  
        System.out.println("Sto leggendo: " + libro.getTitolo() + " di " +  
                           libro.getAutore());  
    }  
}
```

e

```
public interface Programmatore {  
    default void programma(String linguaggio) {  
        System.out.println("Sto programmando in " + linguaggio);  
    }  
}
```

è possibile creare la seguente classe che implementa entrambe le interfacce e ne eredita i metodi:

```
public class ChiStaLeggendo implements Lettore, Programmatore {  
}
```

Ed ecco un esempio di utilizzo:

```
public class TestEreditarietaMultipla {  
    public static void main(String args[]) {  
        ChiStaLeggendo tu = new ChiStaLeggendo();  
        Libro java8 = new Libro("Manuale di Java 8", "Claudio De Sio Cesari");  
        tu.programma("Java");  
        tu.leggi(java8);  
    }  
}
```

E sin qui nessun problema, ma in realtà non sarà sempre così semplice implementare l'ereditarietà multipla. Esistono una serie di casi in cui bisogna conoscere le regole per risolvere i problemi che si presentano. Vediamo le varie regole iniziando dal caso del **Diamond Problem**.

1. Nel caso una classe erediti da due interfacce due metodi di default con la stessa firma (nome più lista di attributi) è necessario che la classe ridefinisca il metodo.

Consideriamo per esempio la seguente interfaccia che dichiara un metodo di default:

```
public interface Solista {
```

```
default void eseguiAssolo() {
//Scala maggiore in DO
System.out.println("DO RE MI FA SOL LA SI");
}
}
```

Estendiamo questa interfaccia due volte con delle specializzazioni che riscrivono il metodo di default:

```
public interface SolistaBlues extends Solista {
default void eseguiAssolo() {
//Scala blues in DO
System.out.println("DO MIb FA SOLb SOL SIb DO");
}
}

public interface SolistaRock extends Solista {
default void eseguiAssolo() {
//Scala pentatonica in DO
System.out.println("DO RE MI SOL LA DO");
}
}
```

Se provassimo ora a creare una classe che implementi entrambe le interfacce, per esempio:

```
public class Chitarrista implements SolistaBlues, SolistaRock {

}
```

Otterremo il seguente errore in compilazione:

```
D:\java8\Codice\modulo_06\esempi\Chitarrista.java:1: error:
class Chitarrista inherits unrelated defaults for
eseguiAssolo() from types SolistaBlues and SolistaRock
public class Chitarrista implements SolistaBlues, SolistaRock {
 ^
1 error
```

Si tratta del famoso “problema dell’ereditarietà a diamante” (in inglese “Diamond Problem”). In realtà dovrebbe essere tradotto come “problema dell’ereditarietà a rombo”. In questo caso infatti, il termine “diamond” dovrebbe essere tradotto come “rombo”, ma è talmente diffusa la traduzione (errata in questo caso) in “diamante” che oramai è uno standard trovare tale nomenclatura. Il termine “rombo” deriva dalla forma che viene fuori dalla rappresentazione grafica dell’ereditarietà multipla in situazioni simili a queste, come si evince dalla Figura 6.1.

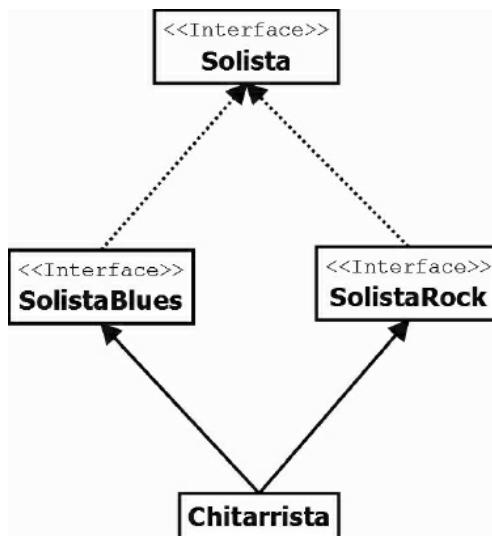


Figura 6.1 - Diamond Problem descritto con UML.

Nell'esempio il compilatore ci ha segnalato un errore, in quanto non è in grado di decidere quale delle due implementazioni ereditate del metodo `eseguiAssolo()` debba essere considerata quella prioritaria.

Si noti che anche se non esistesse l'interfaccia `Solistा` il problema si presenterebbe esattamente allo stesso modo. Diciamo però che per ritrovarci in questa situazione è più probabile che l'interfaccia `Solistा` sia stata definita.

L'unica soluzione che abbiamo per risolvere il conflitto di nomi è quello di ridefinire il metodo per specificarne l'implementazione nuovamente. Per esempio nella seguente classe creiamo un metodo completamente nuovo che usa le implementazioni di entrambe le interfacce implementate:

```

public class Chitarrista implements SolistaBlues, SolistaRock {
    public void eseguiAssolo() {
        //Scala pentatonica + scala blues in DO
        System.out.println("DO RE MI SOL LA DO");
        System.out.println("DO MIB FA SOLB SOL SIB DO");
    }
}
    
```

L'output di questa classe sarà:

```

DO RE MI SOL LA DO
DO MIB FA SOLB SOL SIB DO
    
```

In questo caso però abbiamo duplicato codice già scritto nelle interfacce, ma questo è evitabile utilizzando il reference super. Con la seguente sintassi infatti si ottiene lo stesso risultato:

```
public class Chitarrista implements SolistaBlues, SolistaRock {  
    public void eseguiAssolo() {  
        //Scala pentatonica + scala blues in DO  
        SolistaRock.super.eseguiAssolo();  
        SolistaBlues.super.eseguiAssolo();  
    }  
}
```

La sintassi di `super` in questo caso è diversa da quella che abbiamo visto nel paragrafo 6.5.3. Infatti la semplice chiamata del tipo:

```
super.eseguiAssolo();
```

non avrebbe risolto l'ambiguità. Quindi si usa la sintassi del tipo:

```
NomeSuperClasse.super.nomeMetodoDellaSuperclasse
```

Chiarita la regola 1, ci sono altre possibili situazioni che bisogna gestire quando si implementa l'ereditarietà multipla:

2. nel caso una classe implementi più interfacce, se si presenta all'interno di essa un conflitto di nomi tra un metodo astratto ereditato dalla prima interfaccia ed un metodo concreto ereditato da un'altra interfaccia, dovremo risolvere comunque la situazione ridefinendo il metodo nella classe così come visto nel punto 1.

Nonostante sarebbe stato plausibile ereditare solo il metodo concreto, i progettisti Java hanno optato per l'uniformità. Non interessa come due interfacce paritetiche vadano in conflitto. Se almeno un'interfaccia fornisce un'implementazione al metodo in conflitto, il programmatore deve risolvere l'ambiguità con consapevolezza. In pratica se nell'esempio precedente l'interfaccia `Solistarock` avesse definito come astratto il metodo `eseguiAssolo()`, allora la classe `Chitarrista` avrebbe comunque dovuto riscrivere il metodo come nel caso 1.

3. In una classe che implementa più interfacce, se c'è conflitto di nomi di soli metodi astratti, allora ci troviamo nella situazione pre-Java 8, e la soluzione è quella di ridefinire il metodo nella sotto classe, così come visto per la regola 1.

Se nell'esempio precedente entrambe le interfacce `Solistarock` e `Solistablues` avessero definito come astratti i metodi `eseguiAssolo()`, allora la classe `Chitarrista` avrebbe ereditato un metodo astratto che andava riscritto. Quindi la soluzione descritta nel punto 1 è valida anche in questo caso.

4. Nel caso una classe implementi due interfacce dove una estende l'altra, se nella classe che le implementa c'è conflitto di nomi di metodi ereditati, allora verrà ereditata l'implementazione della interfaccia più specifica.

Ovvero, considerando le interfacce definite nel punto 1, se una nuova classe `ChitarristaRock`

avesse implementato le interfacce `Solistा` e `SolistаRock` come nel seguente esempio:

```
public class ChitarristaRock implements Solista, SolistaRock {  
}
```

allora la classe `ChitarristaRock` avrebbe ereditato l'implementazione dell'interfaccia `SolistаRock`, in quanto più specifica rispetto a quella di `Solistа` (infatti `SolistаRock` aveva ridefinito il metodo di `Solistа`). Quindi l'output della seguente classe di test:

```
public class Concerto {  
    public static void main(String args[]) {  
        Chitarrista chitarrista = new Chitarrista();  
        chitarrista.eseguiAssolo();  
        ChitarristaRock ChitarristaRock = new ChitarristaRock();  
        chitarristaRock.eseguiAssolo();  
    }  
}
```

sarebbe il seguente:

```
DO RE MI SOL LA DO  
DO MIB FA SOLB SOL SIb DO  
DO RE MI SOL LA DO
```

5. Nel caso una classe erediti un conflitto di firme tra un metodo ereditato da una classe estesa e un metodo dell'interfaccia implementata, allora verrà ereditata l'implementazione della classe, anche se questa fosse astratta (questa regola è nota con il nome di “classe vince sempre” in inglese “class always win”).

Per esempio consideriamo la seguente classe astratta:

```
public abstract class ChitarristaAstratto {  
    public abstract void eseguiAssolo();  
}
```

che definisce il metodo `eseguiAssolo()` in maniera astratta. Se provassimo ora a compilare la seguente classe `ChitarristaSolista` che estende la classe `ChitarristaAstratto` e implementa l'interfaccia `Solistа` (che invece definisce un'implementazione di default al metodo `eseguiAssolo()`):

```
public class ChitarristaSolista  
extends ChitarristaAstratto implements Solista {  
}
```

avremmo il seguente errore in compilazione:

```
D:\java8\Codice\modulo_06\esempi\ChitarristaSolista.java:1:  
error: ChitarristaSolista is not abstract and does not  
override abstract method eseguiAssolo() in ChitarristaAstratto  
public class ChitarristaSolista extends ChitarristaAstratto  
implements Solista {  
^  
1 error
```

infatti tra le due definizioni del metodo `eseguiAssolo()` è stata ereditata la versione (benché astratta) della classe `ChitarristaAstratto` piuttosto che la versione concreta dell'interfaccia `Solista` ("class always win"). Se la classe `ChitarristaAstratto` avesse implementato in qualche modo il metodo `eseguiAssolo()` allora la classe `ChitarristaSolista` sarebbe stata compilata senza errori, ereditando il metodo `eseguiAssolo()` della classe `ChitarristaAstratto`.

Riassumendo le 5 regole appena viste: il metodo ereditato si deve sempre riscrivere tranne nei casi 4 e 5 dove vincono rispettivamente l'interfaccia più specifica e la classe.

6.7.5 Differenze tra interfacce e classi astratte

Non è possibile istanziare né classi astratte né interfacce. Inoltre il vantaggio comune che offrono sia le classi astratte sia le interfacce risiede nel fatto che esse possono obbligare le sottoclassi ad implementare comportamenti. Una classe che eredita un metodo astratto infatti, deve fare override del metodo ereditato oppure deve essere dichiarata a sua volta astratta. Dal punto di vista della progettazione quindi, questi strumenti supportano l'astrazione dei dati. Per esempio, ogni veicolo accelera e decelera e quindi tutte le classi non astratte che estendono `Veicolo` devono riscrivere i metodi `accelera()` e `decelera()`.

Un'evidente differenza pratica tra i due concetti è cosa possono definire. Un interfaccia può definire costanti statiche, metodi statici, metodi di default e metodi astratti. Un classe astratta è invece una classe normale, che però non può essere istanziata e può contenere metodi astratti (ma questo non è necessario).

Dunque una classe astratta solitamente non è altro che un'astrazione troppo generica per essere istanziata nel contesto in cui si dichiara. Un buon esempio è la classe `Veicolo`.

```
public abstract class Veicolo {  
    public abstract void accelera();  
    public abstract void decelera();  
}
```

Per esempio, ogni veicolo accelera e decelera e quindi tutte le classi non astratte che estendono `Veicolo` devono riscrivere i metodi `accelera()` e `decelera()`.

```
public class Aereo extends Veicolo {  
    public void accelera() {  
        // override del metodo ereditato  
        . . .  
    }
```

```
public void decelera() {  
    // override del metodo ereditato  
    . . .  
}  
//. . .  
}
```

Un’interfaccia invece, di solito non è una vera astrazione troppo generica per il contesto, semmai è una “astrazione comportamentale”, che non ha senso istanziare nel contesto stesso. Spesso le interfacce hanno nomi che richiamano aggettivi e comportamenti.

Considerando sempre l’esempio del `Veicolo` superclasse di `Aereo`, potremmo introdurre un’interfaccia `Volante` (notare come il nome faccia pensare ad un comportamento più che ad un oggetto astratto) che verrà implementata dalle classi che volano. Ora, se l’interfaccia `Volante` è definita nel seguente modo:

```
public interface Volante {  
    void atterra();  
    void decolla();  
}
```

ogni classe che deve astrarre un concetto di oggetto volante (come un aereo o un uccello) deve implementare l’interfaccia. Riscriviamo quindi la classe `Aereo` nel seguente modo:

```
public class Aereo extends Veicolo implements Volante {  
    public void atterra() {  
        // override del metodo di Volante  
    }  
    public void decolla() {  
        // override del metodo di Volante  
    }  
    public void accelera() {  
        // override del metodo di Veicolo  
    }  
    public void decelera() {  
        // override del metodo di Veicolo  
    }  
}
```

Il vantaggio di programmare con classi astratte ed interfacce verrà spiegato nel prossimo modulo dedicato al polimorfismo.

Riepilogo

Questo modulo è stato dedicato principalmente al secondo paradigma fondamentale dell’Object Orientation: l’**ereditarietà**. È un concetto chiaro semplice da apprendere e l’applicabilità si basa sulla relazione “is a”. Sono stati anche introdotti vari altri argomenti correlati con l’ereditarietà.

Abbiamo visto come il modificatore **final** possa rendere non applicabile l'estensione delle classi o la riscrittura dei metodi ereditati. Abbiamo visto come invece il modificatore **abstract** sia usato per lo scopo opposto, ovvero costringere le sottoclassi ad estendere le classi e a riscrivere i metodi astratti. Abbiamo anche definito i concetti di **specializzazione** e **generalizzazione**. È stato approfondito il concetto di **interfaccia** presentando anche le novità di Java 8 come i **metodi statici**, i **metodi di default** e le **interfacce funzionali**. Obbligatoriamente abbiamo dovuto approfondire il supporto semplificato all'**ereditarietà multipla** e le regole che la governano.

È stato evidenziato che l'ereditarietà e l'incapsulamento coesistono tranquillamente, anzi si convalidano a vicenda. Inoltre l'utilizzo corretto dell'astrazione è la base per non commettere errori che sarebbero amplificati da incapsulamento ed ereditarietà.

Abbiamo anche introdotto il reference **super** con i suoi numerosi utilizzi impliciti (nel caso dei costruttori) o esplicativi. Abbiamo visto come è possibile chiamare metodi di superclassi riscritti nelle sottoclassi. Come è possibile chiamare i costruttori di una superclasse, ed anche come utilizzarlo per risolvere problemi di ambiguità con l'ereditarietà multipla. Infine abbiamo sottolineato alcune **differenze tra classi astratte ed interfacce**, fornendo concetti propedeutici al prossimo modulo.

Esercizi modulo 6

Esercizio 6.a) Object Orientation in Java (teoria), Vero o Falso:

1. L'implementazione dell'ereditarietà implica scrivere sempre qualche riga in meno.
2. La seguente dichiarazione di classe è scorretta:

```
public final class Classe extends AltraClasse {...}
```

3. L'ereditarietà è utile solo se si utilizza la specializzazione. Infatti, specializzando ereditiamo nella sottoclasse (o sottoclassi) membri della superclasse che non bisogna riscrivere. Con la generalizzazione invece creiamo una classe in più, e quindi scriviamo più codice.
4. La parola chiave **super** permette di chiamare metodi e costruttori di superclassi. La parola chiave **this** permette di chiamare metodi e costruttori della stessa classe in cui ci si trova.
5. L'ereditarietà multipla non esiste in Java perché non esiste nella realtà.
6. Un'interfaccia funzionale è un'interfaccia che dichiara un unico metodo di default.
7. Una sottoclasse è più “grande” di una superclasse (nel senso che solitamente aggiunge caratteristiche e funzionalità nuove rispetto alla superclasse).
8. Supponiamo di sviluppare un'applicazione per gestire un torneo di calcio. Esiste ereditarietà derivata da specializzazione tra le classi **Squadra** e **Giocatore**.
9. Supponiamo di sviluppare un'applicazione per gestire un torneo di calcio. Esiste ereditarietà derivata da generalizzazione tra le classi **Squadra** e **Giocatore**.
10. In generale, se avessimo due classi **Padre** e **Figlio**, non esisterebbe ereditarietà tra queste due classi.

Esercizio 6.b)

Data la seguente classe:

```
public class Persona {  
    private String nome;  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
}
```

Commentare la seguente classe `Impiegato` evidenziando dove sono utilizzati i paradigmi object oriented: encapsulamento, ereditarietà e riuso.

```
public class Impiegato extends Persona {  
  
    private int matricola;  
    public void setDati(String nome, int matricola) {  
        setNome(nome);  
        setMatricola(matricola);  
    }  
  
    public void setMatricola(int matricola) {  
        this.matricola = matricola;  
    }  
  
    public int getMatricola() {  
        return matricola;  
    }  
  
    public String dammiDettagli() {  
        return getNome() + ", matricola: " + getMatricola();  
    }  
}
```

Esercizio 6.c) Classi astratte ed interfacce, Vero o Falso:

1. La seguente dichiarazione di classe è scorretta:

```
public abstract final class Classe {...}
```

2. La seguente dichiarazione di classe è scorretta:

```
public abstract class Classe;
```

- 3.** La seguente dichiarazione di interfaccia è scorretta:

```
public final interface Classe {...}
```

- 4.** Una classe astratta contiene per forza metodi astratti.
- 5.** Un'interfaccia può essere estesa da un'altra interfaccia.
- 6.** Una classe può estendere una sola classe ma implementare più interfacce.
- 7.** Il pregio delle classi astratte e delle interfacce è che obbligano le sottoclassi ad implementare i metodi astratti ereditati. Quindi rappresentano un ottimo strumento per la progettazione object oriented.
- 8.** Un interfaccia può dichiarare più costruttori.
- 9.** Un'interfaccia non può dichiarare variabili ma può dichiarare costanti statiche e pubbliche.
- 10.** Una classe astratta può implementare un'interfaccia.

Soluzioni esercizi modulo 6

Esercizio 6.a) Object Orientation in Java (pratica), Vero o Falso:

- 1.** **Falso**, il processo di generalizzazione implica scrivere una classe in più e ciò non sempre implica scrivere qualche riga in meno.
- 2.** **Vero**.
- 3.** **Falso**, anche se dal punto di vista della programmazione la generalizzazione può non farci sempre risparmiare codice, essa ha comunque il pregio di farci gestire le classi in maniera più naturale, favorendo l'astrazione dei dati. Inoltre apre la strada all'implementazione del polimorfismo.
- 4.** **Vero**.
- 5.** **Falso**, l'ereditarietà multipla esiste nella realtà, e in Java esiste in una versione soft perché si eredita solo la parte funzionale.
- 6.** **Falso**, è un'interfaccia che dichiara un unico metodo astratto.
- 7.** **Vero**.
- 8.** **Falso**, una squadra non “è un” giocatore, né un giocatore “è una” squadra. Semmai una squadra “ha un” giocatore ma questa non è la relazione di ereditarietà. Si tratta infatti della relazione di associazione.
- 9.** **Vero**, infatti entrambe le classi potrebbero estendere una classe `Partecipante`.
- 10.** **Falso**, un `Padre` è sempre un `Figlio`, o entrambe potrebbero estendere la classe `Persona`.

Esercizio 6.b)

```
public class Impiegato extends Persona { //Ereditarietà
private int matricola;

public void setDati(String nome, int matricola) {
setNome(nome); //Riuso ed ereditarietà
setMatricola(matricola); //Riuso
}

public void setMatricola(int matricola) {
this.matricola = matricola; //incapsulamento
}

public int getMatricola() {
return matricola; //incapsulamento
}

public String dammiDettagli() {
//Riuso, incapsulamento ed ereditarietà
return getNome() + ", matricola: " + getMatricola();
}
}
```

Esercizio 6.c) Classi astratte ed interfacce, Vero o Falso:

- 1. Vero**, i modificatori `abstract` e `final` sono in contraddizione.
- 2. Vero**, manca il blocco di codice che definisce la classe.
- 3. Vero**, un'interfaccia `final` non ha senso.
- 4. Falso**.
- 5. Vero**.
- 6. Vero**.
- 7. Vero**.
- 8. Falso**, un'interfaccia non può dichiarare costruttori perché non si può istanziare.
- 9. Vero**.
- 10. Vero**.

**Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere l'utilizzo e l'utilità dell'ereditarietà (generalizzazione e specializzazione) (unità 6.1, 6.5)	<input type="checkbox"/>	
Conoscere le conseguenze dell'utilizzo contemporaneo di encapsulamento ed ereditarietà (unità 6.4)	<input type="checkbox"/>	
Comprendere e saper utilizzare i modificatori <code>abstract</code> e <code>final</code> (unità 6.3, 6.6)	<input type="checkbox"/>	
Comprendere e saper utilizzare la parola chiave <code>super</code> (unità 6.5)	<input type="checkbox"/>	
Comprendere e saper utilizzare il nuovo concetto di interfaccia in Java 8 (unità 6.7)	<input type="checkbox"/>	
Sapersi destreggiare nei vari casi complessi dell'ereditarietà multipla (unità 6.7)	<input type="checkbox"/>	

Note:

Polimorfismo

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Comprendere il significato del polimorfismo (unità 7.1).
- ✓ Saper utilizzare l'overload, l'override ed il polimorfismo per dati (unità 7.2 e 7.3).
- ✓ Comprendere e saper utilizzare le collezioni eterogenee, i parametri polimorfi ed i metodi virtuali (unità 7.3).
- ✓ Sapere utilizzare l'operatore `instanceof` ed il casting di oggetti (unità 7.3).

Questo modulo è interamente dedicato al paradigma più complesso dell'Object Orientation: il **polimorfismo**. Si tratta di un argomento vasto ed articolato che viene sfruttato relativamente poco rispetto alla potenza che mette a disposizione dello sviluppatore. Molti programmati Java non riescono neanche a definire questo fondamentale strumento di programmazione. Alla fine di questo modulo invece, il lettore dovrebbe comprenderne pienamente l'importanza. Nel caso non fosse così, potrà sempre tornare a leggere queste pagine in futuro. Una cosa è certa: comprendere il polimorfismo è molto più semplice che implementarlo.

7.1 Polimorfismo

Il **polimorfismo** (dal greco “molte forme”) è un altro concetto che dalla realtà è stato importato nella programmazione ad oggetti. Esso consente di riferirci con un unico termine a “entità” diverse. Ad esempio, sia un telefono fisso sia un portatile permettono di telefonare, dato che entrambi i mezzi sono definibili come telefoni. Telefonare quindi, può essere considerata un’azione polimorfica (ha diverse implementazioni). Il polimorfismo in Java è un argomento complesso che si dirama in vari sottoargomenti. Utilizzando una convenzione con la quale rappresenteremo mediante rettangoli i concetti, e con ovali i concetti che hanno una reale implementazione in Java, cercheremo di schematizzare il polimorfismo e le sue espressioni.

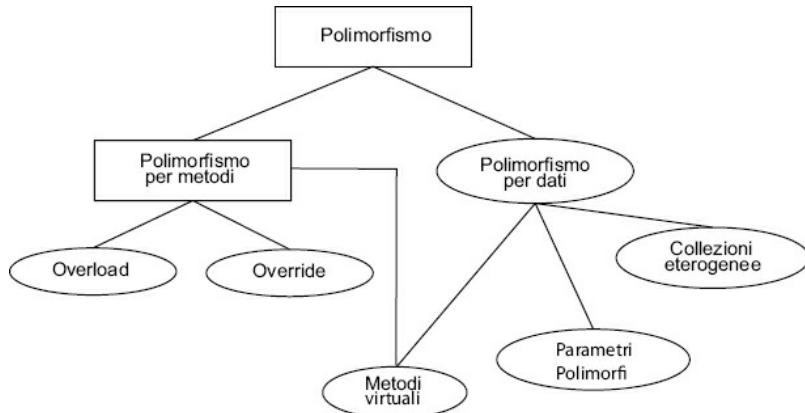


Figura 7.1 - Il polimorfismo in Java.

7.1.1 Convenzione per i reference

Prima di iniziare a definire i vari aspetti del polimorfismo presentiamo una convenzione per la definizione di una variabile di tipo reference. Definire precisamente che cosa sia un reference e come sia rappresentato in memoria non è cosa semplice. Di solito s'identifica un reference con un puntatore (anche se in realtà non è corretto). In molti testi relativi ad altri linguaggi di programmazione, un puntatore viene definito come “una variabile che contiene un indirizzo”. In realtà la definizione di puntatore cambia da piattaforma a piattaforma!

Quindi ancora una volta utilizziamo una convenzione. Possiamo definire un reference come una variabile che contiene due informazioni rilevanti: l'indirizzo in memoria e l'intervallo di puntamento definito dalla relativa classe.

Consideriamo il seguente rifacimento della classe Punto:

```

public class Punto {
private int x;
private int y;
public void setX(int x) {
this.x = x;
}
public void setY(int y) {
this.y = y;
}
public int getX() {
return x;
}
public int getY() {
return y;
}
}

```

Per esempio, se scriviamo:

```
Punto ogg = new Punto();
```

possiamo supporre che il reference `ogg` abbia come indirizzo un valore numerico, ad esempio 10023823, e come intervallo di puntamento `Punto`.

In particolare, ciò che abbiamo definito come intervallo di puntamento farà sì che il reference `ogg` possa accedere all'interfaccia pubblica della classe `Punto`, ovvero a tutti i membri pubblici (`setX()`, `setY()`, `getX()`, `getY()`) dichiarati nella classe `Punto` tramite il reference `ogg`.

L'indirizzo invece farà puntare il reference ad una particolare area di memoria dove risiederà il particolare oggetto istanziato. Viene riportato uno schema rappresentativo in Figura 7.2.

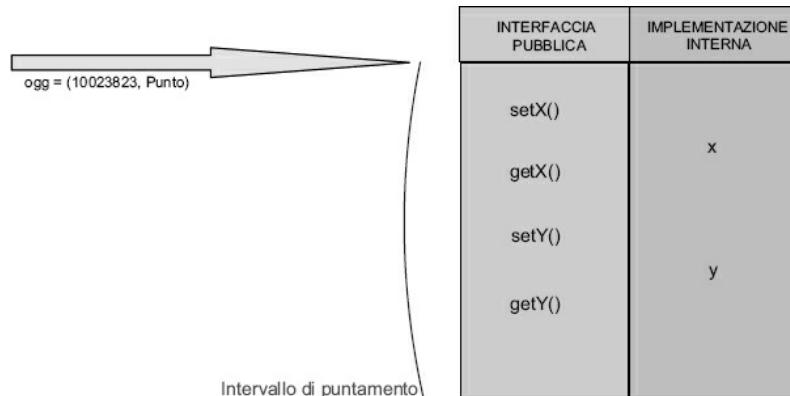


Figura 7.2 - Convenzione per i reference.

7.2 Polimorfismo per metodi

Il polimorfismo per metodi, per quanto asserito sino ad ora, ci permetterà di utilizzare lo stesso nome per metodi differenti. In Java esso trova una sua realizzazione pratica sotto due forme: l'**overload** (che potremmo tradurre con “sovraffunzione”) e l'**override** (che potremmo tradurre con “riscrittura”).

7.2.1 Overload

Definizione 1:

Riguardo un metodo, la coppia costituita dall'identificatore e dalla lista dei parametri è detta **firma** o **segnatura** (dall'inglese *signature*).

In Java un metodo è univocamente determinato non solo dal suo identificatore ma anche dalla sua lista di parametri, cioè dalla sua firma. Quindi, in una classe possono convivere metodi con lo stesso nome ma con differente firma. Su questo semplice concetto si fonda una delle implementazioni più utilizzate di Java: l'overload. Tramite esso il programmatore potrà utilizzare lo stesso nome per metodi diversi. Tutto ciò deve avere un senso logico. Per esempio potremmo assegnare lo stesso

nome a due metodi che concettualmente hanno la stessa funzionalità, ma soddisfano tale funzionalità in maniera differente. Presentiamo di seguito un banale esempio di overload:

```
public class Aritmetica {  
    public int somma(int a, int b) {  
        return a + b;  
    }  
    public float somma(int a, float b) {  
        return a + b;  
    }  
    public float somma(float a, int b) {  
        return a + b;  
    }  
    public int somma(int a, int b, int c) {  
        return a + b + c;  
    }  
    public double somma(int a, double b, int c) {  
        return a + b + c;  
    }  
}
```

In questa classe ci sono ben cinque metodi che hanno lo stesso nome e svolgono somme, ma in modo differente. Se volessimo implementare questi metodi in un altro linguaggio che non supporta l'overload, dovremmo inventare un nome nuovo per ogni metodo. Per esempio il primo di essi si potrebbe chiamare `sommaDueInt()`, il secondo `sommaUnIntEUnFloat()`, il terzo `sommaUnFloatEUnInt()`, il quarto `sommaTreInt()`, il quinto addirittura `sommaUnIntUnDoubleEUnFloat()`! A questo punto l'utilità dell'overload è evidente. Notiamo che la lista dei parametri ha tre criteri di distinzione:

tipale

es.: `somma(int a, int b)` è diverso da `somma(int a, float b)`

numerico

es.: `somma(int a, int b)` è diverso da `somma(int a, int b, int c)`

posizionale

es.: `somma(int a, float b)` è diverso da `somma(float a, int b)`

Gli identificatori che utilizziamo per i parametri invece, non sono criteri di distinzione per i metodi (esempio: `somma(int a, int b)` non è diverso da `somma(int c, int d)`). Inoltre il tipo di ritorno non fa parte della firma di un metodo, quindi non ha importanza per l'implementazione dell'overload.

In alcuni testi l'overload non è considerato aspetto polimorfico di un linguaggio. In questi testi il polimorfismo stesso è definito in maniera diversa da com'è stato definito in questo contesto. Come sempre è tutto relativo all'ambito in cui ci si trova. Se ci

fossimo trovati a discutere di analisi e progettazione object oriented anziché di programmazione, neanche noi avremmo inserito l'overload come argomento del polimorfismo. Se prescindiamo dal linguaggio infatti, ma non è il nostro caso, l'overload non dovrebbe neanche esistere. Inoltre il polimorfismo è stato definito da molti autori come una conseguenza all'implementazione dell'ereditarietà. L'overload e l'ereditarietà in effetti non hanno nulla a che vedere. Se ci limitiamo però a considerare la definizione che abbiamo dato di polimorfismo (stesso nome a cose diverse), l'overload è sicuramente da considerarsi una delle implementazioni del polimorfismo. Qualche autore invece definisce l'overload come "polimorfismo statico" per differenziarlo dall'override (sovrascrittura di metodi nelle sottoclassi) che viene etichettato come "polimorfismo dinamico".

Un altro esempio di overload (che abbiamo già sfruttato inconsapevolmente in questo testo) riguarda il metodo `println()`. Infatti esistono ben dieci metodi `println()` diversi. È possibile passare al metodo non solo stringhe, ma anche interi, booleani, array di caratteri o addirittura `Object`. Il lettore può verificarlo per esercizio consultando la documentazione (suggerimento: `System.out`, è un oggetto della classe `PrintStream` appartenente al package `java.io`).

In realtà avevamo anche visto un altro esempio di overload già nel modulo 2, quando abbiamo introdotto i costruttori, ed anche nel modulo precedente dove abbiamo visto come un costruttore possa invocare un altro costruttore mediante la parola chiave `this`. Ci era parso naturale creare più costruttori in una stessa classe con differente liste di parametri, in realtà stavamo già utilizzando l'overload inconsapevolmente. Segue un semplice esempio:

```
public class Cliente {  
    private String nome;  
    private String indirizzo;  
    private String numeroDiTelefono;  
    public Cliente() {  
        // costruttore inserito esplicitamente (non di default)  
    }  
    public Cliente(String nome) {  
        this.nome = nome;  
    }  
    public Cliente(String nome, String indirizzo) {  
        this(nome);  
        this.indirizzo = indirizzo;  
    }  
    public Cliente(String nome, String indirizzo, String numeroDiTelefono) {  
        this(nome, indirizzo);  
        this.numeroDiTelefono = numeroDiTelefono;  
    }  
    // . . .  
}
```

7.2.2 Varargs

Come già accennato, quando abbiamo presentato il concetto di metodo nel modulo 2, già dalla versione 5 di Java è stata introdotta la possibilità di utilizzare come argomenti dei metodi i cosiddetti **varargs** (abbreviativo per **variable arguments**). Con i varargs è possibile fare in modo che un metodo accetti un numero non precisato di argomenti (compresa la possibilità di passare zero parametri), evitando così la creazione di metodi “overloadati”. La sintassi dei varargs fa uso di **ellissi** (i puntini sospensivi “...”). Per esempio, la seguente implementazione della classe `Aritmetica` potrebbe sostituire con un unico metodo l’overload dell’esempio del paragrafo precedente:

```
public class Aritmetica {  
    public double somma(double... doubles) {  
        double risultato = 0.0D;  
        for (double tmp : doubles) {  
            risultato += tmp;  
        }  
        return risultato;  
    }  
}
```

Infatti, tenendo conto che `ogg` è un oggetto di tipo `Aritmetica`, il seguente codice è valido:

```
System.out.println(ogg.somma(1,2,3));  
System.out.println(ogg.somma());  
System.out.println(ogg.somma(1,2));  
System.out.println(ogg.somma(1,2,3,5,6,8,2,43,4));
```

Segue l’output delle precedenti righe di codice:

```
6.0  
0.0  
3.0  
74.0
```

Il risultato sarà di tipo `double` (a meno di casting).

Effettivamente i varargs all’interno del metodo dove sono dichiarati, sono considerati a tutti gli effetti degli array. Quindi come per gli array se ne può ricavare la dimensione con la variabile `length` e usarli nei cicli. Il vantaggio di avere varargs in luogo di un array o di una collection risiede essenzialmente nel fatto che, per chiamare un metodo che dichiara argomenti variabili, non bisogna creare array o collection.

Quando parliamo di collection, parliamo di un framework definito dalla libreria standard che definisce una serie di classi ed interfacce che rappresentano collezioni di oggetti con caratteristiche ben precise. Per esempio le classi che implementano l’interfaccia `List` hanno la caratteristica di essere ordinate e ogni suo elemento ha un indice, mentre le classi che implementano l’interfaccia `Set` non ammettono duplicati.

La classe probabilmente più usata è l'ArrayList.

Giusto per dare un'idea di cosa stiamo parlando, di seguito mostriamo come si crea e si riempie un ArrayList di stringhe:

```
ArrayList arrayList = new ArrayList();
arrayList.add("ArrayList");
arrayList.add("implementa");
arrayList.add("List");
```

approfondiremo bene il discorso nel modulo relativo alle librerie fondamentali.

Un metodo con varargs viene semplicemente invocato come si fa con un qualsiasi overload. Se per esempio avessimo avuto un array al posto di varargs per il metodo somma():

```
public double somma(double[] doubles) {
    double risultato = 0.0D;
    for (double tmp : doubles) {
        risultato += tmp;
    }
    return risultato;
}
```

per invocare il metodo avremmo dovuto ricorrere alle seguenti righe di codice:

```
double[] doubles = {1.2D, 2, 3.14, 100.0};
System.out.println(ogg.somma(doubles));
```

Insomma, usare i varargs in luogo di array come parametri di un metodo è più comodo.

I varargs danno l'illusione dell'utilizzo dell'overload, anche se parliamo di qualcosa di diverso. Infatti i varargs vanno dichiarati specificando un certo tipo di dato (per esempio `double` o `String`) e il numero di valori da passare è indefinito (e può essere anche zero). Con l'overload possiamo specificare un numero fisso di parametri e di tipi diversi. Inoltre:

1. è possibile dichiarare un unico varargs a metodo;
2. è possibile dichiarare anche altri parametri oltre ad un (unico) varargs a metodo, ma il varargs deve occupare l'ultima posizione tra i parametri.

Per esempio consideriamo di nuovo la classe `Cliente` che definiva un overload di costruttori definito nel paragrafo 7.2.1. Con i varargs potremmo creare un unico costruttore in questo modo:

```
public class ClienteVarargs {
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;
```

```
public ClienteVarargs(String... par) {
    if (par != null) {
        if (par[0] != null) {
            this.nome = par[0];
        }
        if (par[1] != null) {
            this.indirizzo = par[1];
        } if (par[2] != null) {
            this.numeroDiTelefono = par[2];
        }
    }
}
// . .
}
```

Sembra evidente che l'overload rappresenti una soluzione migliore in questo caso.

7.2.3 Override

L'override (e questa volta non ci sono dubbi) è invece considerato un'importantissima caratteristica della programmazione ad oggetti, tanto che qualcuno lo identifica (un po' superficialmente) con il polimorfismo stesso. L'override (che potremmo tradurre con "riscrittura") è il termine object oriented che viene utilizzato per descrivere la caratteristica, che hanno le sottoclassi, di ridefinire un metodo ereditato da una superclasse. Non esisterà override senza ereditarietà. Una sottoclasse è sempre più specifica della classe che estende, e quindi potrebbe ereditare metodi che hanno bisogno di essere ridefiniti per funzionare correttamente nel nuovo contesto.

Ad esempio, supponiamo che una ridefinizione della classe `Punto` (che per convenzione assumiamo bidimensionale) dichiari un metodo `distanzaDall'Origine()` il quale calcola, con la nota espressione geometrica, la distanza tra un punto di determinate coordinate dall'origine degli assi cartesiani. Questo metodo, ereditato all'interno di un'eventuale sottoclasse `PuntoTridimensionale`, ha bisogno di essere ridefinito per calcolare la distanza voluta, tenendo conto anche della terza coordinata. Vediamo quanto appena descritto sotto forma di codice:

```
public class Punto {
    private int x, y;

    public void setX(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
    public void setY(int y) {
        this.y = y;
    }
    public int getY() {
        return y;
    }
}
```

```

}
public double distanzaDallOrigine() {
int tmp = (x*x) + (y*y);
return Math.sqrt(tmp);
}
}

public class PuntoTridimensionale extends Punto {
private int z;

public void setZ(int z) {
this.z = z;
}
public int getZ() {
return z;
}

public double distanzaDallOrigine() {
int tmp = (getX()*getX()) + (getY()*getY())
+ (z*z); // N.B. : x ed y non sono ereditate
return Math.sqrt(tmp);
}
}

```

Per chi non ricorda come si calcola la distanza geometrica tra due punti, se abbiamo i punti P_1 e P_2 tali che hanno coordinate rispettivamente (x_1, y_1) e (x_2, y_2) , la distanza tra essi sarà data da:

$$d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Ora, se P_2 coincide con l'origine, ha coordinate $(0, 0)$ e quindi la distanza tra un punto P_1 e l'origine sarà data da:

$$d(P_1, P_2) = \sqrt{(x_1)^2 + (y_1)^2}$$

Il metodo `sqrt()` della classe `Math` (package `java.lang`) restituisce un valore di tipo `double`, risultato della radice quadrata del parametro passato (il lettore è invitato sempre e comunque a consultare la documentazione). È stato possibile invocarlo con la sintassi `NomeClasse.nomeMetodo` anziché `nomeOggetto.nomeMetodo` perché trattasi di un metodo statico.

Il quadrato di x e di y è stato ottenuto mediante la moltiplicazione del valore per se stesso. Per esercizio il lettore può esplorare la documentazione della classe `Math` per cercare l'eventuale presenza di un metodo per elevare numeri a potenza (suggerimento: "potenza" in inglese si dice "power").

Attenzione! Astrarre un punto bidimensionale creando una classe di nome `Punto` e inserire il metodo `distanzaDallOrigine()` nella stessa classe, sono scelte che

sembrano “accettabili”. Ma non stiamo così violando il paradigma dell’astrazione? La classe non dovrebbe chiamarsi `PuntoBidimensionale`? È giusto che un punto possegga un metodo per calcolare la distanza da un altro punto? I punti non calcolano la distanza nel mondo reale... semmai potrebbe essere utile creare una classe `Righello`! Il discorso continua nell’appendice G on line con un esempio guidato alla programmazione ad oggetti.

Osservare come abbiamo ridefinito il blocco di codice del metodo `distanzaDallOrigine()` nella sottoclasse `PuntoTridimensionale` per introdurre le terza coordinata di cui tenere conto affinché il calcolo della distanza sia eseguito correttamente.

Si noti che abbiamo avuto accesso diretto alla variabile z (senza passare per il metodo “set” o “get”), giacché essa è definita privata nella stessa classe del metodo che la sta usando.

L’override sembra semplice da implementare, ma non è tutto così scontato: ci sono regole da rispettare.

1. Il metodo riscritto nella sottoclasse deve avere la stessa firma (nome e parametri) del metodo della superclasse.
2. Il tipo di ritorno del metodo della sottoclasse deve coincidere con quello del metodo che si sta riscrivendo, o deve essere di un tipo che estende il tipo di ritorno del metodo della superclasse.
3. Il metodo ridefinito nella sottoclasse non deve essere meno accessibile del metodo originale della superclasse.

Per quanto riguarda la prima regola, dando per scontato che il nome del metodo della sottoclasse deve essere identico a quello della superclasse, se parliamo di firme differenti, parliamo di lista di parametri differenti. In casi come questo saremmo di fronte ad un overload in luogo di un override. Infatti non essendo riscritto con la stessa firma, erediteremmo nella sottoclasse anche il metodo della superclasse, e ci sarebbero due metodi con lo stesso nome e differente lista di parametri.

Con la seconda regola risulta invece chiaro che anche il tipo di ritorno è una discriminante (a differenza dell’overload dove invece non lo era). In particolare il tipo di ritorno del metodo può coincidere anche con una sottoclasse del tipo di ritorno del metodo originale. In questo caso si parla di “tipo di ritorno covariante”. Per esempio, sempre considerando il rapporto di ereditarietà che sussiste tra le classi `Punto` e `PuntoTridimensionale`, se nella classe `Punto`, fosse presente il seguente metodo:

```
public Punto elaboraPunto() {  
// . . .  
}
```

allora sarebbe legale implementare il seguente override nella classe `PuntoTridimensionale`:

```
public PuntoTridimensionale elaboraPunto() {
```

```
// . . .
}
```

Infine per la terza regola la situazione è abbastanza chiara. Per esempio, se un metodo ereditato è dichiarato protetto, non si può ridefinire privato, semmai pubblico.

Esiste anche un'altra regola, che sarà trattata nel modulo relativo alle eccezioni.

7.2.4 Annotazione sull'override

Uno degli errori più subdoli che può commettere un programmatore è sbagliare un override. Ovvero ridefinire in maniera non corretta un metodo che si vuole riscrivere in una sottoclasse, magari digitando una lettera minuscola piuttosto che una maiuscola, o sbagliando il numero di parametri in input. In tali casi il compilatore non segnalerà errori, dal momento che non può intuire il tentativo di override in atto. Quindi, per esempio, il seguente codice:

```
public class PuntoTridimensionale {
    public double distanzadallOrigine() { // si scrive distanzaDallOrigine() !
        . . .
    }
}
```

è compilato correttamente, ma non sussisterà nessun override in `MiaClasse`. La complicazione è che il problema si presenterà solo in fase di esecuzione dell'applicazione e non è detto che lo si riesca a correggere velocemente.

Dalla versione 5 di Java è stata introdotta una nuova struttura dati vagamente simile ad una classe, detta **Annotazione**. Un'annotazione permette di *annotare* qualsiasi tipo di componente di un programma Java: dalle variabili ai metodi, dalle classi alle annotazioni stesse. Con il termine *annotare* intendiamo qualificare, marcare. Ovvero, se per esempio annotiamo una classe, consentiremo ad un software (per esempio il compilatore Java) di rendersi conto che tale classe è stata marcata, cosicché potrà implementare un certo comportamento di conseguenza. L'argomento è piuttosto complesso e più avanti troverete un modulo interamente dedicato alle annotazioni. L'esempio più intuitivo di annotazione però, definita dal linguaggio stesso, riguarda proprio il problema di implementazione dell'override che stavamo considerando. Esiste nel package `java.lang` l'annotazione `Override`. Questa la si può (e la si deve) utilizzare per marcare i metodi che vogliono essere override di metodi ereditati. Per esempio:

```
public class PuntoTridimensionale {
    @Override
    public double distanzadallOrigine() {
        . . .
    }
}
```

Notiamo come l'utilizzo di un'annotazione richieda l'uso di un carattere che non avevamo mai

incontrato nella sintassi Java: @.

Marcando con `Override` il nostro metodo, nel caso violassimo una qualche regola dell'override come nel seguente codice:

```
public class PuntoTridimensionale {  
    @Override  
    public double distanzadallOrigine() {  
        . . .  
    }  
}
```

otterremmo un errore direttamente in compilazione:

```
method does not override a method from its superclass  
@Override public double distanzadallOrigine() {  
^  
1 error
```

Avere un errore in compilazione, è molto meglio che averlo in fase di esecuzione.

7.3 Polimorfismo per dati

Il **polimorfismo per dati** permette essenzialmente di poter assegnare un reference di una superclasse ad un'istanza di una sottoclasse. Per esempio, tenendo conto che `PuntoTridimensionale` è sottoclasse di `Punto`, sarà assolutamente legale scrivere:

```
Punto ogg = new PuntoTridimensionale();
```

Il reference `ogg` infatti, punterà ad un indirizzo che valida il suo intervallo di puntamento. Praticamente l'interfaccia pubblica dell'oggetto creato (costituita dai metodi `setX()`, `getX()`, `setY()`, `getY()`, `setZ()`, `getZ()` e `distanzaDallOrigine()`) contiene l'interfaccia pubblica della classe `Punto` (costituita dai metodi `setX()`, `getX()`, `setY()`, `getX()` e `distanzaDallOrigine()`) e così il reference `ogg` "penserà" di puntare ad un oggetto `Punto`. Se volessimo rappresentare graficamente questa situazione potremmo basarci sulla Figura 7.3.

Questo tipo di approccio ai dati ha però una limitazione. Un reference di una superclasse non potrà accedere ai campi dichiarati per la prima volta nella sottoclasse. Nell'esempio otterremmo un errore in compilazione se tentassimo di accedere ai metodi accessor e mutator della terza coordinata `z` del `PuntoTridimensionale` tramite il reference `ogg`. Per esempio, la codifica della seguente riga:

```
ogg.setZ(5);
```

produrrebbe un errore in fase di compilazione, dal momento che `ogg` è un reference che ha un intervallo di puntamento di tipo `Punto`.

A questo punto solitamente il lettore tende a chiedersi: "OK, ho capito cosa significa polimorfismo per dati, ma a che serve?".

Terminata la lettura di questo modulo avremo la risposta.

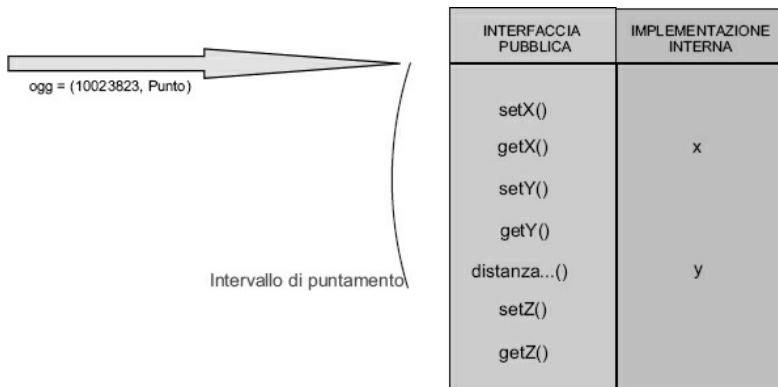


Figura 7.3 – Polimorfismo per dati.

7.3.1 Parametri polimorfi

Sappiamo che i parametri in Java sono sempre passati per valore. Ciò implica che passare un parametro di tipo reference ad un metodo, significa passare il valore numerico del reference, in altre parole il suo indirizzo. A quest'indirizzo potrebbe risiedere un oggetto istanziato da una sottoclasse, grazie al polimorfismo per dati.

In un metodo un parametro di tipo reference si dice **parametro polimorfo** quando, anche essendo di fatto un reference relativo ad una determinata classe, può puntare ad un oggetto istanziato da una sottoclasse. In pratica, sfruttando il polimorfismo per dati, un parametro di un metodo potrebbe in realtà puntare ad oggetti diversi. È il caso del metodo `println()` che prende un parametro di tipo `Object`.

```
Punto p1 = new Punto();
System.out.println(p1);
```

Il lettore ha già appreso precedentemente che tutte le classi (compresa la classe `Punto`) sono sottoclassi di `Object`. Quindi potremmo chiamare il metodo `println()` passandogli come parametro, anziché un'istanza di `Object`, un'istanza di `Punto` come `p1`. Ma a questo tipo di metodo possiamo passare un'istanza di qualsiasi classe, dal momento che vale il polimorfismo per dati ed ogni classe è sottoclasse di `Object`. In particolare il metodo `println()` chiamerà sul parametro passato in input il metodo `toString()` che sicuramente esiste, essendo stato dichiarato nella classe `Object` come vedremo nel paragrafo 7.3.7. Molte classi della libreria standard di Java eseguono un override di questo metodo, restituendo stringhe descrittive dell'oggetto. Se al metodo `println()` passassimo come parametro un oggetto di una classe che non ridefinisce il metodo `toString()`, verrebbe chiamato il metodo `toString()` ereditato dalla classe `Object` (come nel caso della classe `Punto`). Un'implementazione del metodo `toString()` per la classe `Punto` potrebbe essere la seguente:

```
@Override  
public String toString() {  
    return "(" + getX() + "," + getY() + ")";  
}
```

7.3.2 Collezioni eterogenee

Una collezione eterogenea è una collezione composta da oggetti diversi (ad esempio un array di `Object` che in realtà immagazzina oggetti diversi). Anche la possibilità di sfruttare collezioni eterogenee è garantita dal polimorfismo per dati. Infatti un array dichiarato di `Object` potrebbe contenere ogni tipo di oggetto:

```
Object arr[] = new Object[3];  
arr[0] = new Punto(); //arr[0], arr[1], arr[2]  
arr[1] = "Hello World!"; //sono reference ad Object  
arr[2] = new Date(); //che puntano ad oggetti  
                    //istanziati da sottoclassi
```

il che equivale a scrivere:

```
Object arr[] = {new Punto(), "Hello World!", new Date()};
```

Presentiamo di seguito un esempio allo scopo di intuire l'utilità di questi concetti. Per semplicità non incapsuleremo le nostre classi. Immaginiamo di voler realizzare un sistema che stabilisca le paghe dei dipendenti di un'azienda, considerando le seguenti classi:

```
public class Dipendente {  
    public String nome;  
    public int stipendio;  
    public int matricola;  
    public String dataDiNascita;  
    public String dataDiAssunzione;  
}  
  
public class Programmatore extends Dipendente {  
    public String linguaggiConosciuti;  
    public int anniDiEsperienza;  
}  
  
public class Dirigente extends Dipendente {  
    public String orarioDiLavoro;  
}  
  
public class AgenteDiVendita extends Dipendente {  
    public String [] portafoglioClienti;  
    public int provvigioni;  
}
```

Il nostro scopo è realizzare un metodo che stabilisca le paghe dei dipendenti. Potremmo ora utilizzare una collezione eterogenea di dipendenti ed un parametro polimorfo per risolvere il problema in modo semplice, veloce e sufficientemente elegante. Infatti, potremmo dichiarare una collezione eterogenea di dipendenti nel seguente modo:

```
Dipendente [] arr = new Dipendente [180];
arr[0] = new Dirigente();
arr[1] = new Programmatore();
arr[2] = new AgenteDiVendita();
. . .
```

Esiste tra gli operatori di Java un operatore binario che può essere utile in questi contesti: `instanceof`. Tramite esso si può testare a che tipo di oggetto punta in realtà un reference:

```
public void pagaDipendente(Dipendente dip) {
if (dip instanceof Programmatore) {
dip.stipendio = 1200;
}
else if (dip instanceof Dirigente) {
dip.stipendio = 3000;
}
else if (dip instanceof AgenteDiVendita) {
dip.stipendio = 1000;
}
. . .
}
```

Ora possiamo chiamare questo metodo all'interno di un ciclo `foreach` (di 180 iterazioni), passandogli tutti gli elementi della collezione eterogenea, e raggiungere così il nostro scopo:

```
. . .
for (Dipendente dipendente : arr) {
pagaDipendente(dipendente);
}
. . .
```

In particolare l'operatore `instanceof` ha come operandi un reference (il primo operando) e una classe (il secondo operando). Questo operatore restituisce `true` se il primo operando è un reference che punta ad un oggetto istanziato dal secondo operando o ad un oggetto istanziato da una sottoclasse specificata dal secondo operando. Se non si verifica una di queste due condizioni ritorna `false`. Ciò implica che se il metodo `pagaDipendente()` fosse scritto nel seguente modo:

```
public void pagaDipendente(Dipendente dip) {
if (dip instanceof Dipendente) {
dip.stipendio = 1000;
```

```
}
```

```
else if (dip instanceof Programmatore) {
```

```
    . . .
```

tutti i dipendenti sarebbero pagati allo stesso modo. Infatti, anche se `dip` fosse un reference di tipo `Programmatore`, sarebbe comunque verificato il controllo del primo `if`, visto che se `dip` è un `Programmatore`, per l'ereditarietà è anche un `Dipendente`.

Come già accennato più volte, nella libreria standard esiste un nutrito gruppo di classi fondamentali per lo sviluppo, note sotto il nome di “collections”. Si tratta di collezioni eterogenee e ridimensionabili come la classe `ArrayList` vista nella nota che conclude il paragrafo 7.2.2 sui varargs. Le collections rappresentano l'argomento principale del modulo relativo al package `java.util`.

7.3.3 Casting di oggetti

Nell'esempio precedente abbiamo osservato che l'operatore `instanceof` ci permette di testare a quale tipo di istanza punta un reference. Ma già sappiamo che il polimorfismo per dati, fa sì che il reference che punta ad un oggetto istanziato da una sottoclasse non possa accedere ai membri dichiarati nelle sottoclassi stesse. Esiste però la possibilità di ristabilire la piena accessibilità all'oggetto tramite il meccanismo del casting di oggetti.

Rifacciamoci all'esempio appena presentato. Supponiamo che lo stipendio di un programmatore dipenda dal numero di anni di esperienza. In questa situazione, dopo aver testato che il reference `dip` punta ad un'istanza di `Programmatore`, avremo bisogno di accedere alla variabile `anniDiEsperienza`. Se tentassimo di accedervi mediante la sintassi `dip.anniDiEsperienza`, otterremmo sicuramente un errore in compilazione. Ma se utilizzassimo il meccanismo del casting di oggetti supereremmo anche quest'ultimo ostacolo.

In pratica dichiareremo un reference a `Programmatore` e lo faremo puntare all'indirizzo di memoria dove punta il reference `dip`, utilizzando il casting per confermare l'intervento di puntamento. Il nuovo reference, essendo “giusto”, ci permetterà di accedere a qualsiasi membro dell'istanza di `Programmatore`.

Il casting di oggetti sfrutta una sintassi del tutto simile al casting tra dati primitivi:

```
if (dip instanceof Programmatore) {
```

```
    Programmatore pro = (Programmatore) dip;
```

```
    . . .
```

Siamo ora in grado di accedere alla variabile `anniDiEsperienza` tramite la sintassi:

```
    . . .
```

```
    if (pro.anniDiEsperienza > 2)
```

```
    . . .
```

Come al solito cerchiamo di chiarirci le idee cercando di schematizzare la situazione con una

rappresentazione grafica (Figura 7.4).

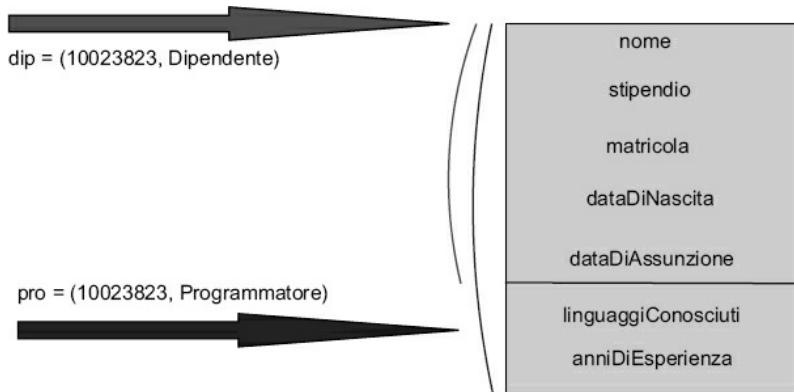


Figura 7.4 – Due diversi tipi di accesso per lo stesso oggetto.

Si noti come i due reference abbiano lo stesso valore numerico (indirizzo) ma differente intervallo di puntamento. Da questo dipende l'accessibilità all'oggetto.

Nel caso tentassimo di assegnare al reference `pro` l'indirizzo di `dip` senza l'utilizzo del casting, otterremmo un errore in compilazione ed un relativo messaggio che ci richiede un casting esplicito. Ancora una volta il comportamento del compilatore conferma la robustezza del linguaggio. Il compilatore non può stabilire se ad un certo indirizzo risiede un determinato oggetto piuttosto che un altro. È solo durante l'esecuzione che Java Virtual Machine può sfruttare l'operatore `instanceof` per risolvere il dubbio.

Il casting di oggetti non si deve considerare come strumento di programmazione standard, piuttosto come un utile mezzo per risolvere problemi progettuali. Una progettazione ideale farebbe a meno del casting di oggetti. Per quanto ci riguarda, all'interno di un progetto la necessità del casting ci porta a pensare ad una "forzatura" e quindi ad un eventuale aggiornamento della progettazione. Tuttavia ci sono casi in cui è inevitabile.

Ancora una volta osserviamo un altro aspetto che fa definire Java come linguaggio semplice da imparare. Il casting è un argomento che esiste anche in altri linguaggi e riguarda i tipi di dati primitivi numerici. Esso viene realizzato troncando i bit in eccedenza di un tipo di dato il cui valore vuole essere forzato ad entrare in un altro tipo di dato "più piccolo". Notiamo che, nel caso di casting di oggetti non viene assolutamente troncato nessun bit, quindi si tratta di un processo completamente diverso! Se però ci astraiamo dai tipi di dati in questione, la differenza non sembra sussistere e Java permette di utilizzare la stessa sintassi, facilitando l'apprendimento e l'utilizzo al programmatore.

7.3.4 Invocazione virtuale dei metodi

Un'invocazione ad un metodo `m` può definirsi **virtuale** quando `m` è definito in una classe `A`, ridefinito in una sottoclasse `B` (override) e invocato su un'istanza di `B`, tramite un reference di `A` (polimorfismo per dati). Quando s'invoca in maniera virtuale il metodo `m`, il compilatore "pensa" di invocare il metodo `m` della classe `A` (virtualmente). In realtà viene invocato il metodo ridefinito nella classe `B`. Un esempio classico è quello del metodo `toString()` della classe `Object`. Abbiamo già accennato al fatto che esso viene sottoposto a override in molte classi della libreria standard. Consideriamo la classe `Date` del package `java.util`. In essa il metodo `toString()` è riscritto in modo tale da restituire informazioni sull'oggetto `Date` (giorno, mese, anno, ora, minuti, secondi, giorno della settimana, ora legale...). Consideriamo il seguente frammento di codice:

```
...
Object obj = new Date();
String s1 = obj.toString();
...
```

Il reference `s1` conterrà la stringa che contiene informazioni riguardo l'oggetto `Date`, unico oggetto istanziato. La situazione è schematizzata in Figura 7.5.

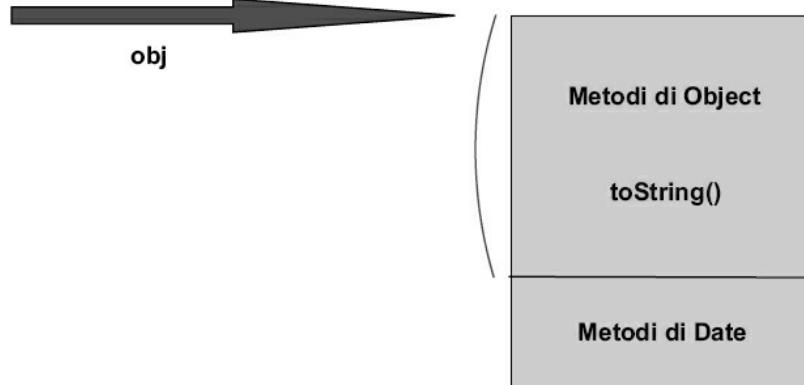


Figura 7.5 - Lo spazio dei metodi di un oggetto Date che sovrascrive `toString()` della classe `Object`.

Il reference `obj` può accedere solamente all'interfaccia pubblica della classe `Object` e quindi anche al metodo `toString()`. Il reference punta però a un'area di memoria dove risiede un oggetto della classe `Date`, nella quale il metodo `toString()` ha una diversa implementazione.

Il metodo `toString()` era già stato chiamato in maniera virtuale nell'esempio del paragrafo sui parametri polimorfi.

7.3.5 Esempio d'utilizzo del polimorfismo

Supponiamo di avere a disposizione le seguenti classi:

```
public abstract class Veicolo {  
    public abstract void accelera();  
    public abstract void decelera();  
}  
  
public class Aereo extends Veicolo {  
    public void decolla() {  
        //...  
    }  
    public void atterra() {  
        //...  
    }  
    @Override  
    public void accelera() {  
        //...  
    }  
    @Override  
    public void decelera() {  
        //...  
    }  
    //...  
}  
  
public class Automobile extends Veicolo {  
    @Override  
    public void accelera() {  
        //...  
    }  
    @Override  
    public void decelera() {  
        //...  
    }  
    public void innestaRetromarcia() {  
        //...  
    }  
    //...  
}  
  
public class Nave extends Veicolo {  
    @Override  
    public void accelera() {  
        //...  
    }  
    @Override  
    public void decelera() {  
        //...  
    }  
    public void gettaAncora() {  
        //...  
    }  
}
```

```
}
```

```
//. . .
```

```
}
```

La superclasse `Veicolo` definisce i metodi `accelera` e `decelera`, che vengono poi ridefiniti in sottoclassi più specifiche quali `Aereo`, `Automobile` e `Nave`.

Si noti che i metodi sono stati dichiarati astratti. Effettivamente, come accelerare un veicolo? Come una nave o come un aereo? La risposta più giusta è che non c'è risposta! È indefinita. Un veicolo sicuramente può accelerare e decelerare ma, in questo contesto, non si può dire come. La soluzione più giusta appare dichiarare i metodi astratti.

Consideriamo la seguente classe, che fa uso dell'overload:

```
public class Viaggiatore {  
    public void viaggia(Automobile a) {  
        a.accelera();  
        //. . .  
    }  
    public void viaggia(Aereo a) {  
        a.accelera();  
        //. . .  
    }  
    public void viaggia(Nave n) {  
        n.accelera();  
        //. . .  
    }  
    //. . .  
}
```

Nonostante l'overload rappresenti una soluzione notevole per la codifica della classe `Viaggiatore`, notiamo una certa ripetizione nei blocchi di codice dei tre metodi. Sfruttando infatti un parametro polimorfo ed un metodo virtuale, la classe `Viaggiatore` potrebbe essere codificata in modo più compatto e funzionale:

```
public class Viaggiatore {  
    public void viaggia(Veicolo v) { //param. Polimorfo  
        v.accelera(); //invocazione metodo virtuale  
        //. . .  
    }  
    //. . .  
}
```

Il seguente frammento di codice infatti, utilizza le precedenti classi:

```
Viaggiatore claudio = new Viaggiatore();
```

```
Automobile fiat500 = new Automobile();
// avremmo potuto istanziare anche una Nave o un Aereo
claudio.viaggia(fiat500);
```

Notiamo la chiarezza e la versatilità del codice. Inoltre ci siamo calati in un contesto altamente estensibile: se volessimo introdurre un nuovo `Veicolo` (supponiamo la classe `Bicicletta`) ci basterebbe codificarla senza toccare quanto scritto finora!

Anche il seguente codice costituisce un valido esempio:

```
Viaggiatore claudio = new Viaggiatore();
Aereo piper = new Aereo();
claudio.viaggia(piper);
```

7.3.6 Polimorfismo e interfacce

- Il polimorfismo per dati funziona anche con le interfacce. Questo significa che possiamo usare reference di interfacce per puntare ad oggetti che implementano queste interfacce. Nel modulo precedente abbiamo asserito che le interfacce possono essere definite per astrarre comportamenti, aggettivi, per poi essere implementati in classi concrete. Nel modulo precedente avevamo definito l'interfaccia `Volante` in questo modo:

```
public interface Volante {
void atterra();
void decolla();
}
```

ogni classe che deve astrarre un concetto di “oggetto volante” (come un aereo, un elicottero o un uccello) deve implementare l'interfaccia. Quindi avevamo riscritto la classe `Aereo` nel seguente modo:

```
public class Aereo extends Veicolo implements Volante {
@Override
public void atterra() {
// override del metodo di Volante
}
@Override
public void decolla() {
// override del metodo di Volante
}
@Override
public void accelera() {
// override del metodo di Veicolo
}
@Override
public void decelera() {
// override del metodo di Veicolo
}
```

}

2. Potremmo quindi creare parametri polimorfi per sfruttare l’interfaccia `Volante`:

```
public class TorreDiControllo {  
    public void autorizzaAtterraggio(Volante v) {  
        v.atterra();  
    }  
  
    public void autorizzaDecollo(Volante v) {  
        v.decolla();  
    }  
}
```

3. E possiamo passare a questi metodi degli “oggetti volanti” creati da classi che implementano l’interfaccia `Volante`.

Riepilogo

In questo modulo abbiamo introdotto alcuni metodi di `Object` come `toString()`, l’operatore `instanceof` e il **casting** di oggetti. Abbiamo soprattutto esplorato il supporto di Java al **polimorfismo** dividendolo in vari sottoargomenti. Il polimorfismo si divide in polimorfismo per metodi e per dati. Il **polimorfismo per metodi** è solo un concetto che trova una sua implementazione in Java tramite **overload** ed **override**. L’**overload** permette di creare metodi con lo stesso nome (ma differente lista dei parametri) nella stessa classe. L’**override** invece permette di riscrivere nelle sottoclassi i metodi ereditati dalle superclassi. Il **polimorfismo per dati** ha di per sé una implementazione in Java e si esprime anche tramite parametri polimorfi e collezioni eterogenee. Inoltre l’utilizzo contemporaneo del polimorfismo per dati e dell’override fa in modo che si possano invocare metodi in maniera virtuale. Tutti gli strumenti di programmazione che abbiamo visto in questo modulo sono molto utili ed un loro corretto utilizzo deve diventare per il lettore uno degli obiettivi fondamentali. Chiunque riesce a “smanettare” con Java ma ci sono altri linguaggi più adatti per “smanettare”. Lo sviluppo Java va accompagnato dalla ricerca di soluzioni di progettazione per agevolare la programmazione. Se non si desidera fare questo forse Java non è il linguaggio giusto da imparare. Non sarà sicuramente semplice utilizzare correttamente i potenti strumenti della programmazione ad oggetti. Nonostante al lettore potrebbero essere chiari tutti gli argomenti che ha studiato sinora, non è detto che sia in grado di creare un programma iniziando da zero, anzi è altamente improbabile se non ha precedenti esperienze. Certamente può aiutare molto conoscere una metodologia object oriented (consultare la bibliografia on line nell’appendice T) o almeno **UML** (a cui sono dedicate le appendici on line I e L). L’esperienza rimane come sempre la migliore palestra. Il consiglio per chi inizia è quindi quello di provare a creare un proprio progetto personale che potrebbe essergli utile, magari non troppo complicato, e non limitarsi ad eseguire gli esercizi forniti dal libro. In questo modo toccherà con mano le difficoltà della progettazione ed anche se il progetto rischia di non vedere la luce un giorno, almeno si inizierà a fare esperienza, si faranno degli errori e si imparerà a correggerli.

Esercizi modulo 7

Esercizio 7.a) Polimorfismo per metodi, Vero o Falso:

1. L'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diverso tipo di ritorno.
2. L'overload di un metodo implica scrivere un altro metodo con nome differente e stessa lista di parametri.
3. La segnatura (o firma) di un metodo è costituita dalla coppia identificatore - lista di parametri.
4. Per sfruttare l'override bisogna che sussista l'ereditarietà.
5. Per sfruttare l'overload bisogna che sussista l'ereditarietà.

Supponiamo che in una classe **B**, la quale estende la classe **A**, ereditiamo il metodo:

```
public int m(int a, String b) { . . . }
```

6. Se nella classe **B** scriviamo il metodo:

```
public int m(int c, String b) { . . . }
```

stiamo facendo overload e non override.

7. Se nella classe **B** scriviamo il metodo:

```
public int m(String a, String b) { . . . }
```

stiamo facendo overload e non override.

8. Se nella classe **B** scriviamo il metodo:

```
public void m(int a, String b) { . . . }
```

otterremo un errore in compilazione.

9. Se nella classe **B** scriviamo il metodo:

```
protected int m(int a, String b) { . . . }
```

otterremo un errore in compilazione.

10. Se nella classe **B** scriviamo il metodo:

```
public int m(String a, int c) { . . . }
```

otterremo un override.

Esercizio 7.b) Polimorfismo per dati, Vero o Falso:

1. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:

```
Veicolo v []= {new Automobile(), new Aereo(), new Veicolo()};
```

2. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:

```
Object o []= {new Veicolo(), new Aereo(), "ciao"};
```

3. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:

```
Aereo a []= {new Veicolo(), new Aereo(), new Aereo()};
```

4. Considerando le classi introdotte in questo modulo, e se il metodo della classe viaggiatore fosse questo:

```
public void viaggia(Object o) {  
    o.accelera();  
}
```

potremmo passargli un oggetto di tipo `Veicolo` senza avere errori in compilazione. Per esempio:

```
claudio.viaggia(new Veicolo());
```

5. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:

```
PuntoTridimensionaleogg = new Punto();
```

6. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:

```
PuntoTridimensionaleogg = (PuntoTridimensionale)new Punto();
```

7. Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:

```
Puntoogg = new PuntoTridimensionale();
```

8. Considerando le classi introdotte in questo modulo, e se la classe `Piper` estende la classe `Aereo`, il seguente frammento di codice non produrrà errori in compilazione:

```
Veicolo a = new Piper();
```

- 9.** Considerando le classi introdotte in questo modulo, il seguente frammento di codice non produrrà errori in compilazione:

```
String stringa = fiat500.toString();
```

- 10.** Considerando le classi introdotte in questo modulo. Il seguente frammento di codice non produrrà errori in compilazione:

```
public void pagaDipendente(Dipendente dip) {  
    if (dip instanceof Dipendente) {  
        dip.stipendio = 1000;  
    }  
    else if (dip instanceof Programmatore) {  
        . . .  
    }  
}
```

Soluzioni esercizi modulo 7

Esercizio 7.a) Polimorfismo per metodi, Vero o Falso:

- 1. Falso.**, l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
- 2. Falso**, l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
- 3. Vero.**
- 4. Vero.**
- 5. Falso**, l'overload di un metodo implica scrivere un altro metodo con lo stesso nome e diversa lista di parametri.
- 6. Falso**, stiamo facendo override. L'unica differenza sta nel nome dell'identificatore di un parametro, che è ininfluente al fine di distinguere metodi.
- 7. Vero**, la lista dei parametri dei due metodi è diversa.
- 8. Vero**, in caso di override il tipo di ritorno non può essere differente.
- 9. Vero**, in caso di override il metodo riscritto non può essere meno accessibile del metodo originale.
- 10. Falso**, otterremo un overload. Infatti, le due liste di parametri differiscono per posizioni.

Esercizio 7.b) Polimorfismo per dati, Vero o Falso:

- 1. Vero.**

2. Vero.

3. Falso, non è possibile inserire in una collezione eterogenea di aerei un `Veicolo` che è superclasse di `Aereo`.

4. Falso, la compilazione fallirebbe già dal momento in cui provassimo a compilare il metodo `viaggia()`. Infatti non è possibile chiamare il metodo `accelera()` con un reference di tipo `Object`.

5. Falso, c'è bisogno di un casting, perché il compilatore non sa a priori il tipo a cui punterà il reference al runtime.

6. Vero.

7. Vero.

8. Vero, infatti `Veicolo` è superclasse di `Piper`.

9. Vero, il metodo `toString()` appartiene a tutte le classi perché ereditato dalla superclasse `Object`.

10. Vero, ma tutti i dipendenti verranno pagati allo stesso modo.

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi

<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere il significato del polimorfismo (unità 7.1)	<input type="checkbox"/>	
Saper utilizzare l'overload, l'override ed il polimorfismo per dati (unità 7.2 e 7.3)	<input type="checkbox"/>	
Comprendere e saper utilizzare le collezioni eterogenee, i parametri polimorfi ed i metodi virtuali (unità 7.3)	<input type="checkbox"/>	
Sapere utilizzare l'operatore <code>instanceof</code> ed il casting di oggetti (unità 7.3)	<input type="checkbox"/>	

Note:

Eccezioni e asserzioni

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Comprendere le differenze tra eccezioni, errori ed asserzioni (unità 8.1).
- ✓ Saper gestire le varie tipologie di eccezioni con i blocchi **try–catch** (unità 8.2, 8.3).
- ✓ Saper utilizzare il costrutto **try with resources** (unità 8.4).
- ✓ Capire e saper utilizzare le asserzioni (unità 8.6).

I concetti relativi ad eccezioni, errori ed asserzioni e le relative gestioni permettono allo sviluppatore di scrivere software robusto, in modo che funzioni correttamente anche in situazioni impreviste. Nelle prossime pagine parleremo non solo di una parte della sintassi fondamentale di Java, ma in qualche modo anche di progettazione. In particolare il meccanismo di propagazione delle eccezioni è una caratteristica del linguaggio da cui non si può prescindere. Le asserzioni invece sono usate relativamente poco, ma in realtà sono un ottimo strumento per scrivere codice funzionante.

8.1 Eccezioni, errori e asserzioni

Dei tre argomenti di cui tratta questo modulo il più importante è sicuramente la gestione delle eccezioni, vero e proprio punto cardine del linguaggio.

Un'**eccezione** è una situazione imprevista che può presentarsi durante il flusso di un'applicazione. È possibile gestirla in Java imparando ad utilizzare cinque semplici parole chiave: `try`, `catch`, `finally`, `throw` e `throws`. Sarà anche possibile creare eccezioni personalizzate e decidere non solo come, ma anche in quale parte del codice gestirle, grazie ad un meccanismo di propagazione estremamente potente. Questo concetto è implementato nella libreria Java mediante la classe `Exception` e le sue sottoclassi. Un esempio di eccezione che potrebbe verificarsi all'interno di un programma è quello relativo ad una divisione tra due variabili numeriche nella quale la variabile divisore ha valore 0. Come è noto, infatti, tale operazione non è eseguibile.

È invece possibile definire un **errore** come una situazione imprevista non dipendente da un errore commesso dallo sviluppatore. A differenza delle eccezioni, quindi, gli errori non sono gestibili. Questo concetto è implementato nella libreria Java mediante la classe `Error` e le sue sottoclassi. Un esempio di errore che potrebbe causare un programma è quello relativo alla terminazione delle risorse di memoria. Questa condizione non è gestibile.

Infine è possibile definire un'**asserzione** come una condizione che deve essere verificata affinché lo sviluppatore consideri corretta una parte di codice. A differenza delle eccezioni e degli errori, le asserzioni rappresentano uno strumento per testare la robustezza del software. Si possono abilitare in

fase di sviluppo e test, ed eventualmente disabilitare in fase di rilascio. Così l'esecuzione del software non subirà alcun tipo di rallentamento. Questo concetto è implementato tramite la parola chiave `assert`. Per esempio un'asserzione potrebbe assicurare che la variabile con funzione di divisore in una divisione debba essere diversa da 0. Se questa condizione non dovesse verificarsi l'esecuzione del codice verrà interrotta.

8.2 Gerarchie e categorizzazioni

Nella libreria standard di Java esiste una gerarchia di classi che mette in relazione la classe `Exception` e la classe `Error`. Entrambe queste classi estendono la superclasse `Throwable`, come si può vedere in Figura 8.1.

Un'ulteriore categorizzazione delle eccezioni è data dalla divisione delle eccezioni in checked e unchecked exception. Ci si riferisce alle `RuntimeException` (e le sue sottoclassi) come unchecked exception. Tutte le altre eccezioni (ovvero tutte quelle che non estendono `RuntimeException`), vengono dette checked exception. Se si utilizza un metodo che lancia una checked exception senza gestirla da qualche parte, la compilazione non andrà a buon fine. Da qui il termine checked exception (in italiano "eccezioni controllate").

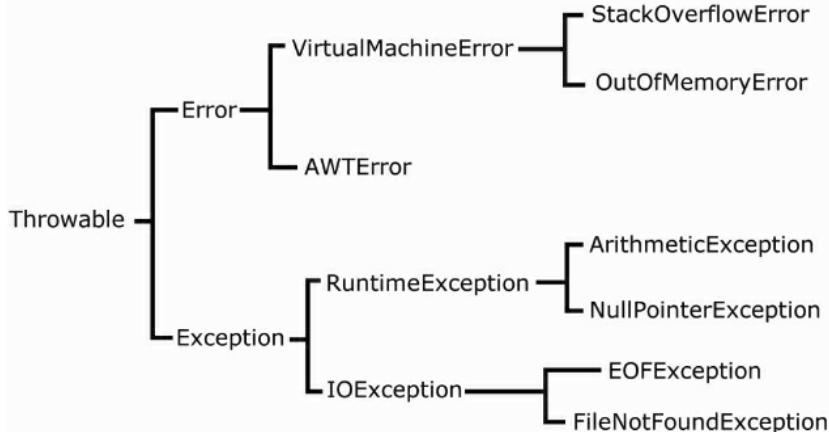


Figura 8.1 - Gerarchia per la classi `Throwable`.

Come abbiamo già detto, non bisognerà fare confusione tra il concetto di errore (problema che un programma non può risolvere) e di eccezione (problema non critico e gestibile). Il fatto che sia la classe `Exception` sia la classe `Error` estendano una classe che si chiama "lanciabile" (`Throwable`) è dovuto al meccanismo con cui la Java Virtual Machine reagisce quando si imbatte in una eccezione-errore. Infatti, se il nostro programma genera un'eccezione durante il runtime, la JVM istanzia un

oggetto dalla classe eccezione relativa al problema e *lancia* l'eccezione appena istanziata (tramite la parola chiave `throw`). Se il nostro codice non *cattura* (tramite la parola chiave `catch`) l'eccezione, il gestore automatico della JVM interromperà il programma generando in output informazioni dettagliate su ciò che è accaduto. Supponiamo che durante l'esecuzione un programma provi ad eseguire una divisione per zero tra interi. La JVM istanzierà un oggetto di tipo `ArithmaticException` (inizializzandolo opportunamente) e lo lancerà. In pratica è come se la JVM eseguisse le seguenti righe di codice:

```
ArithmaticException exc = new ArithmaticException();
throw exc;
```

Tutto avviene dietro le quinte e sarà trasparente allo sviluppatore.

8.3 Meccanismo per la gestione delle eccezioni

Come già asserito in precedenza, lo sviluppatore ha a disposizione alcune parole chiave per gestire le eccezioni: `try`, `catch`, `finally`, `throw` e `throws`. Se è necessario sviluppare una parte di codice che potenzialmente potrebbe scatenare un'eccezione, è possibile circondarla con un blocco `try` seguito da uno o più blocchi `catch`. Per esempio:

```
public class Ecc1 {
public static void main(String args[]) {
int a = 10;
int b = 0;
int c = a/b;
System.out.println(c);
}
}
```

Questa classe sarà compilata senza problemi ma genererà un'eccezione durante la sua esecuzione, dovuta all'impossibilità di eseguire una divisione per zero. In tal caso la JVM, dopo aver interrotto il programma, produrrà il seguente output:

```
Exception in thread "main" java.lang.ArithmaticException: / by zero at
Ecc1.main(Ecc1.java:6)
```

Un messaggio di sicuro molto esplicativo dal momento che sono stati evidenziati:

- ❑ il tipo di eccezione (`java.lang.ArithmaticException`);
- ❑ un messaggio descrittivo (`/ by zero`);
- ❑ il metodo in cui è stata lanciata l'eccezione (`at Ecc1.main`);
- ❑ il file in cui è stata lanciata l'eccezione (`Ecc1.java`);
- ❑ la riga in cui è stata lanciata l'eccezione (`:6`).

L'unico problema è che il programma è terminato prematuramente. Utilizzando le parole chiave `try` e `catch` sarà possibile gestire l'eccezione in maniera personalizzata:

```
public class Ecc2 {  
    public static void main(String args[]) {  
        int a = 10;  
        int b = 0;  
        try {  
            int c = a/b;  
            System.out.println(c);  
        }  
        catch (ArithmaticException exc) {  
            System.out.println("Divisione per zero...");  
        }  
    }  
}
```

Quando la JVM eseguirà tale codice incontrerà la divisione per zero della prima riga del blocco `try` e lancerà l'eccezione `ArithmaticException`, che verrà catturata nel blocco `catch` seguente. Quindi non sarà eseguita la riga che doveva stampare la variabile `c`, bensì la stringa "Divisione per zero..." con la quale abbiamo gestito l'eccezione ed abbiamo permesso al nostro programma di terminare in maniera naturale. Come il lettore avrà sicuramente notato la sintassi dei blocchi `try - catch` è piuttosto strana, ma presto ci si farà l'abitudine in quanto la si ritroverà più volte in tutti i programmi Java. In particolare il blocco `catch` deve dichiarare un parametro (come se fosse un metodo) del tipo dell'eccezione che deve essere catturata. Nell'esempio precedente il reference `exc` puntava proprio all'eccezione che la JVM aveva istanziato e lanciato. Infatti, tramite esso è possibile reperire informazioni proprio sull'eccezione stessa. Il modo più utilizzato e semplice per ottenere informazioni su ciò che è successo è invocare il metodo `printStackTrace()` sull'eccezione:

```
int a = 10;  
int b = 0;  
try {  
    int c = a/b;  
    System.out.println(c);  
}  
catch (ArithmaticException exc) {  
    exc.printStackTrace();  
}
```

Il metodo `printStackTrace()` produrrà in output i messaggi informativi di cui sopra che il programma avrebbe prodotto se l'eccezione non fosse stata gestita, ma senza interrompere il programma stesso.

È fondamentale che si dichiari, tramite il blocco `catch`, un'eccezione del tipo giusto. Per esempio, il seguente frammento di codice:

```
int a = 10;  
int b = 0;
```

```
try {
    int c = a/b;
    System.out.println(c);
}
catch (NullPointerException exc) {
    exc.printStackTrace();
}
```

produrrebbe un'eccezione non gestita e quindi un'immediata terminazione del programma. Infatti il blocco `try` non ha mai lanciato una `NullPointerException`, ma una `ArithmetricException`.

Come per i metodi, anche per i blocchi `catch` i parametri possono essere polimorfi. Per esempio, il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

contiene un blocco `catch` che gestirebbe qualsiasi tipo di eccezione, essendo `Exception` la superclasse da cui discende ogni altra eccezione. Il reference `exc` è in questo esempio un parametro polimorfo.

È anche possibile far seguire ad un blocco `try` più blocchi `catch`, come nel seguente esempio:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmetricException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

In questo modo il nostro programma risulterebbe più robusto e gestirebbe diversi tipi di eccezioni. Nel peggior dei casi (ovvero se il blocco `try` lanciasse un'eccezione non prevista) l'ultimo blocco `catch` gestirebbe il problema.

È fondamentale l'ordine dei blocchi `catch`. Se avessimo:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    exc.printStackTrace();
}
catch (ArithmetricException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
```

gli ultimi due `catch` sarebbero superflui e il compilatore segnalerebbe l'errore nel seguente modo:

```
C:\Ecc2.java:12: exception java.lang.ArithmetricException has already been
caught catch (ArithmetricException exc) {
^
C:\Ecc2.java:15: exception java.lang.NullPointerException has already been
caught catch (NullPointerException exc) {
^
2 errors
```

Ci sono casi in cui il contenuto dei blocchi `catch` è identico, nonostante servano per catturare eccezioni diverse, per esempio:

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmetricException exc) {
    System.out.println(exc.getMessage());
}
catch (NullPointerException exc) {
    System.out.println(exc.getMessage());
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

Per evitare ridondanze è possibile dichiarare in un unico `catch`, più di un parametro come mostrato di seguito:

```
int a = 10;
int b = 0;
try {
int c = a/b;
System.out.println(c);
}
catch (ArithmetricException | NullPointerException exc) {
System.out.println(exc.getMessage());
}
catch (Exception exc) {
exc.printStackTrace();
}
```

Il simbolo “|” viene utilizzato per separare i nomi delle classi eccezione.

La possibilità di catturare più eccezioni in un unico blocco `catch` è stata introdotta solo nella versione 7 del linguaggio.

È possibile far seguire ad un blocco `try`, oltre a blocchi `catch`, un altro blocco definito dalla parola chiave `finally`, per esempio:

```
public class Ecc4 {
public static void main(String args[]) {
int a = 10;
int b = 0;
try {
int c = a/b;
System.out.println(c);
}
catch (ArithmetricException exc) {
System.out.println("Divisione per zero...");
}
catch (Exception exc) {
exc.printStackTrace();
}
finally {
System.out.println("Operazione terminata");
}
}
}
```

Ciò che è definito in un blocco `finally` viene eseguito in qualsiasi caso, sia se viene lanciata l'eccezione sia se non viene lanciata. Per esempio, è possibile utilizzare un blocco `finally` quando esistono operazioni critiche che devono essere eseguite in qualsiasi caso. L'output del precedente

programma è:

```
Divisione per zero...
Operazione terminata
```

Se invece la variabile `b` fosse impostata a `2` piuttosto che a `0`, l'output sarebbe:

```
5
Operazione terminata
```

Un classico esempio (più significativo del precedente) in cui la parola `finally` è spesso utilizzata è il seguente:

```
public void selectFromDB() {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        conn = DriverManager.getConnection(url, username, password);
        stmt = conn.createStatement();
        rs = stmt.executeQuery("SELECT * FROM PERSONA");
        while (rs.next()) {
            System.out.println(rs.getString(1));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        rs = null;
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        stmt = null;
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
conn = null;  
}  
}  
}
```

Il metodo precedente tenta di eseguire una `SELECT` verso un database, tramite le interfacce JDBC offerte dal package `java.sql`. Nell'esempio `stmt` è un oggetto di tipo `Statement` e `conn` è di tipo `Connection`. Il comando `executeQuery()` specifica una banale query. Se ci sono problemi (per esempio sintassi SQL scorretta, chiave primaria già presente etc.) la JVM lancerà una `SQLException` che verrà catturata nel relativo blocco `catch`. In ogni caso, dopo il tentativo di interrogazione, la connessione al database deve essere chiusa, così come anche l'oggetto `Statement` e l'oggetto `ResultSet` (per maggiori approfondimenti su JDBC rimandiamo il lettore al modulo relativo oppure per una breve introduzione all'indirizzo <http://www.claudiodesio.com/java/jdbc.htm>).

È possibile anche far seguire direttamente ad un blocco `try` un blocco `finally`. Quest'ultimo verrà eseguito sicuramente dopo l'esecuzione del blocco `try`, sia se l'eccezione sarà lanciata sia se non sarà lanciata. Comunque, se l'eccezione fosse lanciata, non essendo gestita con un blocco `catch`, il programma terminerebbe anormalmente.

8.4 Try with resources

Dalla versione 7 di Java, alcune classi ed interfacce (tra cui `Connection`) sono state riviste per supportare il meccanismo del cosiddetto **try with resources**. Questo permette la chiusura automatica degli oggetti che necessiterebbero di essere chiusi, una volta utilizzati. La sintassi prevede la dichiarazione dell'oggetto (o degli oggetti) da chiudere automaticamente come parametri del blocco `try`. Segue un esempio (equivalente al precedente):

```
try(Connection conn = DriverManager.getConnection(url, username, password);  
    Statement stmt = conn.createStatement();  
    ResultSet rs = stmt.executeQuery("SELECT * FROM PERSONA ")) {  
    while (rs.next()) {  
        System.out.println(rs.getString(1));  
    }  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

Gli oggetti *chiudibili* `conn`, `stmt` e `rs`, sono quindi dichiarati come se fossero parametri del blocco `try` (che per l'occasione dovrebbe essere chiamato blocco *try with resources*). Quando il blocco terminerà la sua esecuzione, le tre risorse verranno automaticamente chiuse. Questo avverrà sia nel caso positivo (il codice viene eseguito correttamente) sia nel caso negativo (per esempio viene lanciata un'eccezione). Esattamente come se i comandi per chiudere le risorse si trovassero all'interno di una clausola `finally`. Si tratta di un vantaggio non da poco.

Anche con un blocco try with resources è possibile dichiarare un numero arbitrario di blocchi catch ed un finally, come se si trattasse di un try ordinario. Questi blocchi verranno eseguiti solo dopo che le risorse saranno state chiuse.

Se durante l'esecuzione del blocco `try` venisse lanciata un'eccezione, questa verrà considerata prioritaria rispetto ad eventuali eccezioni lanciate automaticamente dal blocco `try with resources` (nel caso si presentasse qualche problema nel chiudere le risorse utilizzate). In tal caso la JVM lancerà l'eccezione applicativa e dichiarerà **suppressed** le eccezioni lanciate durante la chiusura automatica degli oggetti utilizzati (ricordiamo che tali oggetti verranno chiusi come se si trovassero in clausola `finally`). Queste eccezioni saranno comunque impostate come attributi dell'eccezione applicativa lanciata e sarà possibile accedervi chiamando il metodo `getSuppressed()` sull'oggetto eccezione che viene lanciato.

L'utilizzo di questo costrutto è permesso utilizzando tutte le classi che implementano l'interfaccia `AutoCloseable` o l'interfaccia `closeable`. Incontreremo più avanti altri esempi di questo costrutto.

8.5 Eccezioni personalizzate e propagazione dell'eccezione

Il lettore con meno esperienza di programmazione potrebbe chiedersi in questo momento perché utilizzare il meccanismo delle eccezioni piuttosto che delle semplici condizioni `if`. Nei prossimi paragrafi cercheremo di capire le differenze.

Ci sono alcune tipologie di eccezioni che sono più frequenti e quindi più conosciute dagli sviluppatori Java. Si tratta di:

- ❑ `NullPointerException`: probabilmente la più frequente tra le eccezioni. Viene lanciata dalla JVM quando per esempio viene chiamato un metodo su di un reference che invece punta a `null`.
- ❑ `ArrayIndexOutOfBoundsException`: questa eccezione viene lanciata quando si prova ad accedere ad un indice di un array troppo alto o minore di zero.
- ❑ `ClassCastException`: eccezione particolarmente insidiosa. Viene lanciata al runtime quando si prova ad effettuare un cast verso un tipo di classe sbagliato.

Queste eccezioni appartengono tutte al package `java.lang`. Inoltre, se si utilizzano altri package come `java.io`, bisognerà gestire spesso le eccezioni come `IOException` e le sue sottoclassi (`FileNotFoundException`, `EOFException` etc.). Stesso discorso con la libreria `java.sql` e l'eccezione `SQLException`, il package `java.net` e la `ConnectException` e così via. Lo sviluppatore imparerà con l'esperienza come gestire tutte queste eccezioni.

È però altrettanto probabile che qualche volta occorra definire nuovi tipi di eccezione. Infatti, per un particolare programma, potrebbe essere una eccezione anche una divisione per 5. Più verosimilmente, un programma che deve gestire in maniera automatica le prenotazioni per un teatro potrebbe voler lanciare un'eccezione nel momento in cui si tenti di prenotare un posto non più disponibile. In tal caso la soluzione è estendere la classe `Exception` ed eventualmente aggiungere

membri ed effettuare override di metodi come `toString()`. Segue un esempio:

```
public class PrenotazioneException extends Exception {  
    public PrenotazioneException() {  
        // Il costruttore di Exception chiamato inizializza la  
        // variabile privata message  
        super("Problema con la prenotazione");  
    }  
    public String toString() {  
        return getMessage() + ": posti esauriti!";  
    }  
}
```

La “nostra” eccezione contiene informazioni sul problema e rappresenta un’astrazione corretta. Tuttavia la JVM non può lanciare automaticamente una `PrenotazioneException` nel caso si tenti di prenotare quando non ci sono più posti disponibili. La JVM, infatti, sa quando lanciare una `ArithmetricException` ma non sa quando lanciare una `PrenotazioneException`. In tal caso sarà compito dello sviluppatore lanciare l’eccezione. Esiste infatti la parola chiave `throw` (in inglese *lancia*) che permette il lancio di un’eccezione tramite la seguente sintassi:

```
PrenotazioneException exc = new PrenotazioneException();  
throw exc;
```

o equivalentemente (dato che il reference `exc` poi non sarebbe più utilizzabile):

```
throw new PrenotazioneException();
```

Il lancio dell’eccezione dovrebbe seguire un controllo condizionale come il seguente:

```
if (postiDisponibili == 0) {  
    throw new PrenotazioneException();  
}
```

Il codice precedente farebbe terminare prematuramente il programma a meno di gestire l’eccezione come segue:

```
try {  
    //controllo sulla disponibilità dei posti  
    if (postiDisponibili == 0) {  
        //lancio dell'eccezione  
        throw new PrenotazioneException();  
    }  
    //istruzione eseguita  
    // se non viene lanciata l'eccezione  
    postiDisponibili--;  
}  
catch (PrenotazioneException exc){
```

```
System.out.println(exc.toString());
}
```

Il lettore avrà sicuramente notato che il codice precedente non rappresenta un buon esempio di gestione dell'eccezione: dovendo utilizzare la condizione `if`, sembra infatti superfluo l'utilizzo dell'eccezione. In effetti è così! Ma ci deve essere una ragione per la quale esiste la possibilità di creare eccezioni personalizzate e di poterle lanciare. Questa ragione è la “propagazione dell'eccezione” per i metodi chiamanti. La potenza della gestione delle eccezioni è dovuta essenzialmente a questo meccanismo di propagazione. Per comprenderlo bene affidiamoci ad un esempio. Supponiamo di avere la seguente classe:

```
public class Botteghino {
private int postiDisponibili;

public Botteghino() {
postiDisponibili = 100;
}

public void prenota() {
try {
//controllo sulla disponibilità dei posti
if (postiDisponibili == 0) {
    //lancio dell'eccezione
throw new PrenotazioneException();
}
//metodo che realizza la prenotazione
// se non viene lanciata l'eccezione
postiDisponibili--;
}
catch (PrenotazioneException exc) {
System.out.println(exc.toString());
}
}
}
```

La classe `Botteghino` astrae in maniera semplicistica un botteghino virtuale che permette di prenotare i posti in un teatro. Ora consideriamo la seguente classe eseguibile (con metodo `main()`) che utilizza la classe `Botteghino`:

```
public class GestorePrenotazioni {
public static void main(String [] args) {
Botteghino botteghino = new Botteghino();
for (int i = 1; i <= 101; ++i){
botteghino.prenota();
System.out.println("Prenotato posto n° " + i);
}
}
}
```

Per una classe del genere, il fatto che l'eccezione sia gestita all'interno della classe `Botteghino` rappresenta un problema. Infatti l'output del programma sarà:

```
Prenotato posto n° 1
Prenotato posto n° 2
...
Prenotato posto n° 99
Prenotato posto n° 100
Problema con la prenotazione: posti esauriti!
Prenotato posto n° 101
```

C'è però una contraddizione. Gestire eccezioni è sempre un'operazione da compiere, ma non sempre bisogna gestire eccezioni laddove si presentano. In questo caso l'ideale sarebbe gestire l'eccezione nella classe `GestorePrenotazioni` piuttosto che nella classe `Botteghino`:

```
public class GestorePrenotazioni {
    public static void main(String [] args) {
        Botteghino botteghino = new Botteghino();
        try {
            for (int i = 1; i <= 101; ++i) {
                botteghino.prenota();
                System.out.println("Prenotato posto n° " + i);
            }
        } catch (PrenotazioneException exc) {
            System.out.println(exc.toString());
        }
    }
}
```

Tutto ciò è fattibile grazie al meccanismo di propagazione dell'eccezione di Java. Per compilare la classe `Botteghino` però, non basta rimuovere il blocco `try-catch` dal metodo `prenota()`, ma bisogna anche utilizzare la parola chiave `throws` nel seguente modo:

```
public void prenota() throws PrenotazioneException {
    //controllo sulla disponibilità dei posti
    if (postiDisponibili == 0) {
        //lancio dell'eccezione
        throw new PrenotazioneException();
    }
    //metodo che realizza la prenotazione
    // se non viene lanciata l'eccezione
    postiDisponibili--;
}
```

In questo modo otterremo il seguente desiderabile output:

```
Prenotato posto n° 1  
Prenotato posto n° 2  
...  
Prenotato posto n°99  
Prenotato posto n°100  
Problema con la prenotazione: posti esauriti!
```

Se non utilizzassimo la clausola `throws` nella dichiarazione del metodo, il compilatore non compilerebbe il codice precedente. Segnalerebbe che il metodo `prenota()` potrebbe lanciare l'eccezione `PrenotazioneException` (il quale è evidente al compilatore per la parola chiave `throw`) e che questa non viene gestita. In particolare il messaggio di errore restituito sarebbe simile al seguente:

```
Botteghino.java:5: unreported exception PrenotazioneException; must be caught  
or declared to be thrown
```

Questo messaggio è una ulteriore prova delle caratteristiche di robustezza di Java.

Con la clausola `throws` nella dichiarazione del metodo è come se avvertissimo il compilatore che siamo consapevoli che il metodo possa lanciare al runtime la `PrenotazioneException` e di non preoccuparsi in quanto gestiremo in un'altra parte del codice l'eccezione.

Se un metodo chiamante volesse utilizzare un altro metodo `daChiamare` che dichiara con una clausola `throws` il possibile lancio di un certo tipo di eccezione, il metodo chiamante dovrebbe gestire l'eccezione con un blocco `try-catch` che include la chiamata al metodo `daChiamare`, oppure dovrebbe dichiarare anch'esso una clausola `throws` alla stessa eccezione. Ad esempio, questo vale per il metodo `main()` della classe `GestorePrenotazioni`.

Molti metodi della libreria standard sono dichiarati con clausola `throws` a qualche eccezione. Per esempio numerosi metodi delle classi del package `java.io` dichiarano clausole `throws` alla `IOException` (eccezione di input - output). Appare ancora più chiara ora la categorizzazione tra eccezioni checked e unchecked: le checked exception devono essere per forza gestite per poter compilare, mentre le unchecked no, dato che si presentano solo al runtime.

È possibile dichiarare nella clausola `throws` anche più di un'eccezione, separando le varie tipologie con virgole, come nel seguente esempio:

```
public void prenota() throws PrenotazioneException, NullPointerException { . . . }
```

Prima dell'avvento della versione 7, il seguente codice:

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

non era compilabile. Nonostante fosse evidente che il metodo in questione rilanciasse una delle due eccezioni `FirstException` o `SecondException`, avendo utilizzato un blocco `catch` che rilancia una `Exception`, bisognava sostituire (o quantomeno aggiungere) nella clausola `throws` anche `Exception`. Java 7 ha eliminato questo comportamento introducendo un'analisi a livello di compilazione che verifica la veridicità della clausola `throws`.

Già dalla versione 1.4 del linguaggio è stata introdotta una nuova caratteristica alle eccezioni. La classe `Throwable` fu modificata per supportare un semplice meccanismo di wrapping. Spesso infatti, si rende necessario per lo sviluppatore catturare una certa eccezione per lanciarne un'altra. Per esempio:

```
try {
...
} catch(UnaCertaEccezione e) {
throw new UnAltraEccezione();
}
```

In tali casi però, l'informazione della prima eccezione (nell'esempio `UnaCertaEccezione`) viene persa. Si era quindi costretti a creare un'eccezione personalizzata che poteva contenere un'altra come variabile d'istanza. Per esempio:

```
public class WrapperException {
private Exception altraEccezione;
public WrapperException(Exception altraEccezione) {
this.setAltraEccezione(altraEccezione);
}
public void setAltraEccezione(Exception altraEccezione) {
this.altraEccezione = altraEccezione;
}
public Exception getAltraEccezione() {
```

```
    return altraEccezione;
}
//...
}
```

Con questo tipo di eccezione è possibile includere un'eccezione in un'altra, così:

```
try {
...
} catch(UnaCertaEccezione e) {
throw new WrapperException(e);
}
```

Ma dalla versione 1.4 non è più necessario creare un'eccezione personalizzata per utilizzare eccezioni wrapper. Nella classe `Throwable` sono stati introdotti i metodi `getCause()` e `initCause(Throwable)` e i due nuovi costruttori `Throwable(Throwable)` e `Throwable(String, Throwable)`. È quindi ora possibile per esempio codificare le seguenti istruzioni:

```
try {
...
} catch(ArithmeticException e) {
throw new SecurityException(e);
}
```

È inoltre possibile concatenare un numero arbitrario di eccezioni.

8.5.1 Warnings

Quando compiliamo un file Java, il compilatore oltre a compilare correttamente o dare errori, potrebbe anche restituire dei warning (in italiano “avvertimento”). Per esempio, compilando la classe `PrenotazioneException` (e in generale una qualsiasi classe che estende `Throwable`) con un IDE qualsiasi (Eclipse, Netbeans o anche EJE) otterremo il seguente output:

```
D:\java8\Codice\modulo_08\esempi\PrenotazioneException.java:1:
warning: [serial] serializable class PrenotazioneException has no
definition of serialVersionUID
public class PrenotazioneException extends Exception {
^
1 warning
```

Non si tratta di un errore, la classe è stata compilata e il file `.class` è stato creato. Il compilatore ci sta solo avvertendo che questo file è stato definito in modo potenzialmente pericoloso. In particolare ci è stato segnalato che la nostra classe è *serializzabile* e non ha definito un `serialVersionUID`. Arrivati a questo punto del libro non abbiamo ancora le nozioni necessarie a comprendere esattamente il significato di questo warning, tuttavia una spiegazione almeno parziale è dovuta (anche se sarà tutto più chiaro dopo aver affrontato l'argomento della serializzazione). La classe `Throwable`

implementa un'interfaccia denominata `Serializable` (in italiano *serializzabile*). Questa è definita nel package `java.io` della libreria standard. È un'interfaccia che non dichiara metodi ma che, se implementata, consente agli oggetti che la implementano di essere *serializzati*. Un oggetto serializzabile per esempio può essere immagazzinato in un file o fatto viaggiare in rete. Ora non ci dilungheremo nei tecnicismi della serializzazione (che sarà affrontata approfonditamente nel modulo relativo all'Input-Output) visto che il nostro obiettivo è solo quello di capire il significato di questo warning. Se una classe è serializzabile ci viene chiesto dal compilatore di definire un attributo denominato `serialVersionUID`. Questo potrebbe rendersi necessario nel caso la nostra classe dovesse essere usata in ambienti distribuiti (ovvero nel caso gli oggetti della classe dovessero essere usati da virtual machine diverse viaggiando attraverso la rete). In casi come questi dovrebbe esserci una copia della classe serializzabile su macchine diverse. Questo comporta che senza la dichiarazione esplicita del `serialVersionUID` a garantire che le due versioni siano compatibili, la lettura dell'oggetto serializzabile potrebbe fallire e potrebbe essere lanciata una `InvalidClassException`. Per risolvere questo warning è possibile dichiarare il `serialVersionUID` per esempio nel seguente modo:

```
private static final long serialVersionUID = 8144963013726442881L;
```

Si noti che è possibile specificare un qualsiasi modificatore d'accesso, anche se solitamente si preferisce `private`, visto che non c'è ragione di esporre all'esterno questo attributo. È obbligatorio invece che si tratti di un `long` dichiarato `static` e `final`. Il valore da assegnare a `serialVersionUID` è arbitrario, sarebbe bene generarne uno univoco come nell'esempio, ma anche il valore `1L` andrebbe bene! Gli IDE come Eclipse e Netbeans posseggono delle facilities che permettono la generazione automatica di questi valori.

Un'altra possibilità per risolvere il warning, valida solo nel caso si sia sicuri che questa classe non debba essere serializzata, è quella di utilizzare l'annotazione `SuppressWarnings` sulla classe stessa nel seguente modo:

```
@SuppressWarnings("serial")
public class PrenotazioneException extends Exception {
```

In questo caso è come se avessimo raccomandato al compilatore di non preoccuparsi perché il rischio è controllato.

Nel caso invece compilate direttamente da riga di comando nel seguente modo:

```
javac PrenotazioneException.java
```

Allora nessun warning verrà lanciato. Infatti, affinché sia segnalato il nostro warning, bisogna compilare con l'opzione `-Xlint:all` (o semplicemente con `-Xlint`):

```
javac -Xlint:all PrenotazioneException.java
```

È anche possibile specificare quale tipo di warning deve essere segnalato, per esempio con il seguente comando:

```
javac -Xlint:serial PrenotazioneException.java
```

verranno segnalati solo warning sul `serialVersionUID` mancante.

Per maggiori informazioni sulle opzioni del compilatore, è possibile consultare la documentazione dei tool del JDK.

In generale anche dai vari IDE dovrebbe essere possibile impostare un'opzione affinché il compilatore non segnali dei warning.

Per quanto riguarda EJE bisogna andare nel menù File ⇒ Opzioni ⇒ Tab Java e deselezionare il check box relativo ai warnings.

8.5.2 Precisazione sull'override

Quando si fa override di un metodo non è possibile specificare clausole `throws` su eccezioni che il metodo base non comprende nella propria clausola `throws`. È comunque possibile da parte del metodo che effettua override dichiarare una clausola `throws` ad eccezioni che sono sottotipi di eccezioni comprese dal metodo base nella propria clausola `throws`. Per esempio:

```
public class ClasseBase {  
    public void metodo() throws java.io.IOException {}  
  
    class SottoClasseCorretta1 extends ClasseBase {  
        public void metodo() throws java.io.IOException {}  
    }  
  
    class SottoClasseCorretta2 extends ClasseBase {  
        public void metodo() throws java.io.FileNotFoundException {}  
    }  
  
    class SottoClasseCorretta3 extends ClasseBase {  
        public void metodo() {}  
    }  
  
    class SottoClasseScorretta extends ClasseBase {  
        public void metodo() throws java.sql.SQLException {}  
    }
```

La classe `ClasseBase` comprende un metodo che dichiara nella sua clausola `throws` una `IOException`. La classe `SottoClasseCorretta1` effettua override del metodo e dichiara la stessa `IOException` nella sua clausola `throws`. La classe `sottoClasseCorretta2` fa override del metodo e dichiara una `FileNotFoundException`, che è sottoclasse di `IOException` nella sua clausola `throws`. La classe `SottoClasseCorretta3` fa override del metodo e non dichiara clausole `throws`. Infine la classe `SottoClasseScorretta` fa override del metodo e dichiara una `SQLException` nella sua clausola `throws`, e ciò è illegale.

8.6 Introduzione alle asserzioni

Nella versione 1.4 di Java fu introdotta una clamorosa caratteristica al linguaggio. Clamorosa perché si è dovuto modificare la lista delle parole chiave con una nuova: `assert`, cosa mai accaduta nelle precedenti major release. Un'asserzione è un'istruzione che permette di testare eventuali comportamenti previsti in un'applicazione. Ogni asserzione richiede sia verificata un'espressione booleana che lo sviluppatore ritiene vada verificata nel punto in cui viene dichiarata. Se la verifica è negativa si deve parlare di bug. Le asserzioni possono quindi rappresentare un'utile strumento per accertarsi che il codice scritto si comporti come ci si aspetta. Lo sviluppatore può disseminare il codice di asserzioni in modo da testare la robustezza del codice in maniera semplice ed efficace. Lo sviluppatore può infine disabilitare la lettura delle asserzioni da parte della JVM in fase di rilascio del software, così che l'esecuzione non venga rallentata. Moltissimi sviluppatori pensano che l'utilizzo delle asserzioni sia una delle tecniche di maggior successo per scopare bug. Eppure le asserzioni sono relativamente poco usate nell'ambito della programmazione Java. Forse per qualcuno sono fuori moda, ma la realtà è che ai più non è molto chiaro come debbano essere utilizzate. Inoltre, le asserzioni rappresentano anche un ottimo strumento per documentare il comportamento interno di un programma, favorendo la manutenibilità dello stesso.

Esistono due tipi di sintassi per poter utilizzare le asserzioni:

1. `assert espressione_booleana;`
2. `assert espressione_booleana: espressione_stampabile;`

Con la sintassi 1) quando l'applicazione esegue l'asserzione valuta l'`espressione_booleana`. Se questa è `true` il programma prosegue normalmente, ma se il valore è `false` viene lanciato l'errore `AssertionError`. Per esempio l'istruzione:

```
assert b > 0;
```

è semanticamente equivalente a:

```
if (!(b>0)) {  
    throw new AssertionError();  
}
```

A parte l'eleganza e la compattezza del costrutto `assert`, la differenza tra le precedenti due espressioni è notevole. Le asserzioni rappresentano, più che un'istruzione applicativa classica, uno strumento per testare la veridicità delle assunzioni formulate dallo sviluppatore sulla propria applicazione. Se la condizione che viene asserita dal programmatore è falsa, l'applicazione terminerà immediatamente mostrando le ragioni tramite uno stack-trace (metodo `printStackTrace()` di cui sopra). Infatti si è verificato qualcosa che non era previsto dallo sviluppatore stesso. È possibile disabilitare la lettura delle asserzioni da parte della JVM una volta rilasciato il proprio prodotto, al fine di non rallentarne l'esecuzione. Ciò evidenzia la differenza tra le asserzioni e tutte le altre istruzioni applicative.

Rispetto alla sintassi 1), la sintassi 2) permette di specificare anche un messaggio esplicativo tramite l'`espressione_stampabile`. Per esempio

```
assert b > 0: b;
```

oppure:

```
assert b > 0: "il valore di b è " + b;
```

o:

```
assert b > 0: getMessage();
```

o anche:

```
assert b > 0: "assert b > 0 = " + (b > 0);
```

L'espressione_stampabile può essere una qualsiasi espressione che ritorni un qualche valore (quindi non è possibile invocare un metodo con tipo di ritorno `void`). La sintassi 2) permette dunque di migliorare lo stack-trace delle asserzioni.

Il tool JUnit è probabilmente il più famoso software per eseguire test automatici. Tra gli autori figurano due mostri sacri dell'informatica come Kent Beck, autore del processo Test Driven Development e della metodologia Extreme Programming, ed Erich Gamma, storico co-autore del primo libro sui Design Pattern. In particolare JUnit è stato creato per creare “unit test” (in italiano “test unitari”) ovvero creare software che testi automaticamente ognuna delle classi che creiamo. Il grande vantaggio di questo approccio è che alla fine di una giornata lavorativa (ma anche all'inizio o durante) sarà possibile testare la correttezza delle nostre classe con un semplice clic, ed ottenere un eventuale report dei bachi di regressioni del nostro progetto creati con lo sviluppo giornaliero. JUnit è completamente basato su asserzioni ed esiste da prima che le asserzioni fossero introdotte in Java. Infatti gli autori crearono dei metodi, tipo `assertEquals()` o `assertTrue()`, per verificare la correttezza del nostro codice. Per maggiori informazioni su JUnit, consultare l'indirizzo <http://junit.org/>.

8.6.1 Progettazione per contratto

Il meccanismo delle asserzioni deve il suo successo ad una tecnica di progettazione nota con il nome di *Progettazione per contratto* (Design by contract) sviluppata da Bertrand Meyer. Tale tecnica è una caratteristica fondamentale del linguaggio di programmazione sviluppato da Meyer stesso: l'Eiffel (per informazioni <http://www.eiffel.com>). Ma è possibile progettare per contratto più o meno agevolmente con qualsiasi linguaggio di programmazione. La tecnica si basa in particolare su tre tipologie di asserzioni: precondizioni, postcondizioni ed invarianti (le invarianti a loro volta si dividono in interne, sul flusso di esecuzione etc.).

Con una **precondizione** lo sviluppatore può specificare quale deve essere lo stato

dell'applicazione nel momento in cui viene invocata un'operazione. In questo modo si rende esplicito chi ha la responsabilità di testare la correttezza dei dati. L'utilizzo dell'asserzione riduce sia il pericolo di dimenticare completamente il controllo sia quello di effettuare troppi controlli (perché si possono abilitare e disabilitare). Dal momento che si tende ad utilizzare le asserzioni in fase di test e debugging, non bisogna mai confondere l'utilizzo delle asserzioni con quello della gestione delle eccezioni. Nel paragrafo 8.6.3 verranno esplicite regole da seguire per l'utilizzo delle asserzioni.

Con una **postcondizione** lo sviluppatore può specificare quale deve essere lo stato dell'applicazione nel momento in cui un'operazione viene completata. Le postcondizioni rappresentano un modo utile per dire che cosa fare senza dire come. In altre parole è un altro metodo per separare interfaccia ed implementazione interna.

È infine utilizzabile il concetto di **invariante**, che se applicato ad una classe permette di specificare vincoli per tutti gli oggetti istanziati. Questi possono trovarsi in uno stato che non rispetta il vincolo specificato (detto *stato inconsistente*) solo temporaneamente durante l'esecuzione di qualche metodo, al termine del quale lo stato deve ritornare *consistente*.

La progettazione per contratto è appunto una tecnica di progettazione e non di programmazione. Essa consente per esempio di testare la consistenza dell'ereditarietà. Una sottoclasse infatti, potrebbe indebolire le precondizioni e fortificare postcondizioni e invarianti di classe al fine di convalidare l'estensione. Al lettore interessato ad approfondire le sue conoscenze sulla progettazione per contratto consigliamo di dare uno sguardo alla bibliografia on line nell'appendice T.

Se per caso la vostra esigenza è quella di usare codice preesistente creato con una versione di Java precedente alla versione 1.4, dovete consultare l'appendice H.

8.6.2 Note per l'esecuzione di programmi che utilizzano la parola assert

Come più volte detto, è possibile in fase di esecuzione abilitare o disabilitare le asserzioni. Come al solito bisogna utilizzare flag, questa volta applicandoli al comando `java`, ovvero `-enableassertions` (o più brevemente `-ea`) per abilitare le asserzioni, e `-disableassertions` (o `-da`) per disabilitarle. Per esempio:

```
java -ea MioProgrammaConAsserzioni
```

abilita da parte della JVM la lettura dei costrutti `assert`. Mentre:

```
java -da MioProgrammaConAsserzioni
```

disabilita le asserzioni in modo tale da non rallentare l'applicazione. Siccome le asserzioni sono di default disabilitate, il precedente codice è esattamente equivalente al seguente:

```
java MioProgrammaConAsserzioni
```

Sia per l'abilitazione sia per la disabilitazione valgono le seguenti regole:

1. se non si specificano argomenti dopo i flag di abilitazione o disabilitazione delle asserzioni, verranno abilitate o disabilitate le asserzioni in tutte le classi del nostro programma (ma non nelle classi della libreria standard utilizzate). Questo è il caso dei precedenti esempi.
2. Specificando invece il nome di un package seguito da tre puntini, si abilitano o si disabilitano le asserzioni in quel package e in tutti i sottopackage. Per esempio il comando:

```
java -ea -da:miopackage... MioProgramma
```

abiliterà le asserzioni in tutte le classi tranne quelle del package `miopackage`, specificando solo i tre puntini, invece, si abilitano o si disabilitano le asserzioni nel package di default (ovvero la cartella da dove parte il comando);

3. specificando solo un nome di una classe, si abilitano o si disabilitano le asserzioni in quella classe. Per esempio il comando:

```
java -ea:... -da:MiaClasse MioProgramma
```

abiliterà le asserzioni in tutte le classi del package di default, tranne che nella classe `MiaClasse`.

4. È anche possibile abilitare o disabilitare le asserzioni delle classi della libreria standard che si vuole utilizzare mediante i flag `-enablesystemassertions` (o più brevemente `-esa`) e `-disablesystemassertions` (o `-dsa`). Anche per questi flag valgono le regole di cui sopra.

Per quanto riguarda la fase di esecuzione non esistono differenze sul come sfruttare le asserzioni tra la versione 1.4 e le versioni successive.

In alcuni programmi critici è possibile che lo sviluppatore si voglia assicurare che le asserzioni siano abilitate. Con il seguente blocco di codice statico:

```
static {  
    boolean assertsEnabled = false;  
    assert assertsEnabled = true;  
    if (!assertsEnabled)  
        throw new RuntimeException("Asserts must be enabled!");  
}
```

è possibile garantire che il programma sia eseguibile solo se le asserzioni sono abilitate. Il blocco infatti, prima dichiara ed inizializza la variabile booleana `assertsEnabled` a `false`, per poi cambiare il suo valore a `true` se le asserzioni sono abilitate. Se le asserzioni non sono abilitate il programma termina con il lancio della `RuntimeException`, altrimenti continua. Ricordiamo che il blocco statico viene eseguito (cfr. modulo 5) un'unica volta nel momento in cui la classe che lo contiene viene

caricata. Per questa ragione il blocco statico dovrebbe essere inserito nella classe del main(), così da essere sicuri di ottenere il risultato voluto.

8.6.3 Quando usare le asserzioni

Non tutti gli sviluppatori possono essere interessati all'utilizzo delle asserzioni. Un'asserzione non può ridursi a un modo conciso di esprimere una condizione regolare. Un'asserzione è invece il concetto fondamentale di una metodologia di progettazione tesa a rendere i programmi più robusti. Nel momento in cui però lo sviluppatore decide di utilizzare tale strumento, dovrebbe essere suo interesse utilizzarlo correttamente. I seguenti consigli derivano dall'esperienza e dallo studio dell'autore sui testi relativi alle asserzioni.

Consiglio 1)

È spesso consigliato (anche nella documentazione ufficiale) di non utilizzare precondizioni per testare la correttezza dei parametri di metodi pubblici. È invece raccomandato l'utilizzo delle precondizioni per testare la correttezza dei parametri di metodi privati, protetti o con visibilità a livello di package. Questo dipende dal fatto che un metodo non pubblico ha la possibilità di essere chiamato da un contesto limitato, corretto e funzionante. Ciò implica la presunzione che le nostre chiamate al metodo in questione siano corrette, ed è quindi lecito rinforzare tale concetto con un'asserzione. Supponiamo di avere un metodo con visibilità di package come il seguente:

```
public class InstancesFactory {  
    Object getInstance(int index) {  
        assert (index == 1 || index == 2);  
        switch (index) {  
            case 1:  
                return new Instance1();  
            case 2:  
                return new Instance2();  
        }  
    }  
}
```

La classe precedente implementa una soluzione personalizzata basata sul pattern denominato *Factory Method* (per informazioni sul concetto di pattern, cfr. appendice D).

Se questo metodo può essere chiamato solo da classi che appartengono allo stesso package della classe `InstancesFactory`, non deve mai accadere che il parametro `index` sia diverso da 1 o 2, perché tale situazione rappresenterebbe un bug.

Se invece il metodo `getInstance()`, fosse dichiarato `public`, la situazione sarebbe diversa. Infatti, un eventuale controllo del parametro `index` dovrebbe essere considerato ordinario, e quindi da gestire magari mediante il lancio di un'eccezione:

```
public class InstancesFactory {
```

```
public Object getInstance(int index) throws Exception {
if (!(index == 1 || index == 2)) {
throw new Exception("Indice errato: " + index);
}
switch (index) {
case 1:
return new Instance1();
case 2:
return new Instance2();
}
}
```

L'uso di un'asserzione, in tal caso, non garantirebbe la robustezza del programma, ma solo la sua eventuale interruzione se fossero abilitate le asserzioni al runtime, non potendo a priori controllare la chiamata al metodo. In pratica una precondizione di questo tipo violerebbe il concetto object oriented di metodo pubblico.

Consiglio 2)

È sconsigliato l'utilizzo di asserzioni laddove si vuole testare la correttezza di dati che sono inseriti da un utente. Le asserzioni dovrebbero testare la consistenza del programma con se stesso, non la consistenza dell'utente con il programma. L'eventuale input non corretto da parte di un utente è giusto che sia gestito mediante eccezioni, non asserzioni. Per esempio, modifichiamo la classe Data di cui abbiamo parlato nel modulo 5 per spiegare l'incapsulamento:

```
public class Data {
private int giorno;
. .
public void setGiorno(int g) {
assert (g > 0 && g <= 31): "Giorno non valido";
giorno = g;
}
. . .
```

dove il parametro `g` del metodo `setGiorno()` veniva passato da un utente mediante un oggetto interfaccia che rappresentava un'interfaccia grafica (*codice 5.2 bis*):

```
. .
Data unaData = new Data();
unaData.setGiorno(interfaccia.dammiGiornoInserito());
unaData.setMese(interfaccia.dammiMeseInserito());
unaData.setAnno(interfaccia.dammiAnnoInserito());
. . .
```

Come il lettore avrà intuito, l'utilizzo della parola chiave `assert` non è corretto. Infatti, nel caso le

asserzioni fossero abilitate in fase di esecuzione dell'applicazione e l'utente inserisse un valore errato per inizializzare la variabile giorno, l'applicazione si interromperebbe con un `AssertionError`. Se le asserzioni non fossero abilitate, nessun controllo impedirebbe all'utente di inserire valori errati. La soluzione ideale sarebbe gestire la situazione tramite un'eccezione, per esempio:

```
public void setGiorno(int g) throws RuntimeException {
    if (!(g > 0 && g <= 31)) {
        throw new RuntimeException("Giorno non valido");
    }
    giorno = g;
}
```

In questo caso specifico la condizione è ampiamente migliorabile.

Consiglio 3)

L'uso delle asserzioni invece, ben si adatta alle postcondizioni ed alle invarianti. Per **postcondizione** intendiamo una condizione che viene verificata appena prima che termini l'esecuzione di un metodo (ultima istruzione). Segue un esempio:

```
public class Connection {
    private boolean isOpen = false;
    public void open() {
        // ...
        isOpen = true;
        // ...
        assert isOpen;
    }
    public void close() throws ConnectionException {
        if (!isOpen) {
            throw new ConnectionException(
                "Impossibile chiudere connessioni non aperte!");
        }
        // ...
        isOpen = false;
        // ...
        assert !isOpen;
    }
}
```

Dividiamo le **invarianti in interne, di classe e sul flusso di esecuzione**.

Per **invarianti interne** intendiamo asserzioni che testano la correttezza dei flussi del nostro codice. Per esempio il seguente blocco di codice:

```
if (i == 0) {
    ...
} else if (i == 1) {
```

```
    . . .
} else { // ma sicuramente (i == 2)
    . . .
}
```

può diventare più robusto con l'uso di un'asserzione:

```
if (i == 0) {
    . . .
} else if (i == 1) {
    . . .
} else {
assert i == 2 : "Attenzione i = " + i + "!";
    . . .
}
```

Un tale tipo di invariante viene usato con più probabilità all'interno di una clausola `default` di un costrutto `switch`. Spesso lo sviluppatore sottovaluta il costrutto omettendo la clausola `default`, perché suppone che il flusso passi sicuramente per un certo `case`. Le asserzioni sono molto utili per convalidare le nostre supposizioni. Per esempio, il seguente blocco di codice:

```
switch(tipoAuto) {
case Auto.SPORTIVA:
    . . .
break;
case Auto.LUSSO:
    . . .
break;
case Auto.UTILITARIA:
    . . .
break;
}
```

può diventare più robusto con l'uso di un'asserzione:

```
switch(tipoAuto) {
case Auto.SPORTIVA:
    . . .
break;
case Auto.LUSSO:
    . . .
break;
case Auto.UTILITARIA:
    . . .
break;
default:
assert false : "Tipo auto non previsto : " + tipoAuto;
}
```

Per **invarianti di classe** intendiamo particolari invarianti interne che devono essere vere per tutte le istanze di una certa classe, in ogni momento del loro ciclo di vita, tranne che durante l'esecuzione di alcuni metodi. All'inizio ed al termine di ogni metodo però, lo stato dell'oggetto deve tornare *consistente*. Per esempio, un oggetto della seguente classe:

```
public class Bilancia {  
    private double peso;  
    public Bilancia() {  
        azzeraLancetta();  
        assert lancettaAzzerata(); // invariante di classe  
    }  
    private void setPeso(double grammi) {  
        assert grammi > 0; // pre-condizione  
        peso = grammi;  
    }  
    private double getPeso() {  
        return peso;  
    }  
    public void pesa(double grammi) {  
        if (grammi <= 0) {  
            throw new RuntimeException("Grammi <= 0! ");  
        }  
        setPeso(grammi);  
        mostraPeso();  
        azzeraLancetta();  
        assert lancettaAzzerata(); // invariante di classe  
    }  
    private void mostraPeso() {  
        System.out.println("Il peso è di " + peso + " grammi");  
    }  
    private void azzeraLancetta() {  
        setPeso(0);  
    }  
    private boolean lancettaAzzerata () {  
        return peso == 0;  
    }  
}
```

potrebbe, dopo ogni pesatura, azzerare la lancetta (notare che solo i due metodi pubblici terminano con un'asserzione).

Per **invarianti sul flusso di esecuzione** intendiamo asserzioni posizionate in zone del codice che non dovrebbero mai essere raggiunte. Per esempio, se abbiamo codice commentato in questo modo:

```
public void metodo() {  
    if (flag == true) {  
        return;  
    }  
    // L'esecuzione non dovrebbe mai arrivare qui!
```

}

potremmo sostituire il commento con un asserzione sicuramente false:

```
public void metodo() {  
    if (flag == true) {  
        return;  
    }  
    assert false;  
}
```

8.6.4 Conclusioni

In questo modulo abbiamo raggruppato argomenti che possono sembrare simili e ne abbiamo esplicitato le differenze. Il concetto di fondo è che l'utilizzo della gestione delle eccezioni è fondamentale per la creazione di applicazioni robuste. Le asserzioni invece rappresentano un comodo meccanismo per testare la robustezza delle nostre applicazioni. La gestione delle eccezioni non è consigliabile, è obbligatoria! Le asserzioni invece sono molto spesso ingiustamente snobbate. In effetti bisogna avere un po' di esperienza per sfrutarne a pieno le potenzialità. Infatti, la progettazione per contratto è un argomento complesso che va studiato a fondo per poter ottenere risultati corretti. Ciononostante anche l'utilizzo dei concetti più semplici come le postcondizioni può migliorare la robustezza delle nostre applicazioni.

Riepilogo

In questo modulo abbiamo dapprima distinto i concetti di **eccezione**, **errore** ed **asserzione**. Poi abbiamo categorizzato le eccezioni e gli errori con una panoramica sulle classi principali. Inoltre abbiamo ulteriormente suddiviso le tipologie di eccezioni in **checked** ed **unchecked**. È stato presentato il meccanismo alla base della gestione delle eccezioni, parallelamente alle cinque parole chiave che ne permettono la gestione. I blocchi **try-catch-finally** consentono di gestire localmente le eccezioni. Le coppie **throw-throws** supportano invece la propagazione (in maniera robusta) delle eccezioni. Abbiamo anche mostrato come creare eccezioni personalizzate. La possibilità di astrarre il concetto di eccezione con gli oggetti e la possibilità di sfruttare il meccanismo di call-stack (propagazione dell'errore) permettono di creare applicazioni contemporaneamente object-oriented, semplici e robuste. Abbiamo visto anche come il costrutto **try with resources** ci garantisca semplicità e robustezza in alcune occasioni.

Le asserzioni hanno la caratteristica di potere essere abilitate o disabilitate al momento dell'esecuzione del programma, e a proposito abbiamo esplicitato ogni tipo di opzione che deve essere usata da riga di comando. Abbiamo anche introdotto il loro utilizzo all'interno della progettazione per contratto, introducendo i concetti di **precondizioni**, **postcondizioni** ed **invarianti**. Infine sono stati dati alcuni consigli sui casi in cui è opportuno utilizzare le asserzioni.

Esercizi modulo 8

Esercizio 8.a) Gestione delle eccezioni e degli errori, Vero o Falso:

1. Ogni eccezione che estende una `ArithmeticException` è una unchecked exception.
2. Un `Error` si differenzia da una `Exception` perché non può essere lanciato, infatti non estende la classe `Throwable`.
3. Il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
int c = a/b;
System.out.println(c);
}
catch (ArithmeticException exc) {
System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
System.out.println("Reference nullo...");
}
catch (Exception exc) {
System.out.println("Eccezione generica...");
}
finally {
System.out.println("Finally!");
}
```

produrrà il seguente output:

```
Divisione per zero...
Eccezione generica...
Finally!
```

4. Il seguente frammento di codice:

```
int a = 10;
int b = 0;
try {
int c = a/b;
System.out.println(c);
}
catch (Exception exc) {
System.out.println("Eccezione generica...");
}
catch (ArithmeticException exc) {
System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
```

```
System.out.println("Reference nullo...");  
}  
finally {  
    System.out.println("Finally!");  
}
```

produrrà un errore al runtime.

5. La parola chiave `throw` permette di lanciare “a mano” solo le sottoclassi di `Exception` che crea il programmatore.
6. La parola chiave `throw` permette di lanciare “a mano” solo le sottoclassi di `Exception`.
7. Se un metodo fa uso della parola chiave `throw`, affinché la compilazione abbia buon esito, nello stesso metodo deve essere gestita l’eccezione che si vuole lanciare oppure il metodo stesso deve utilizzare una clausola `throws`.
8. Non è possibile estendere la classe `Error`.
9. Se un metodo `m2` fa override di un altro metodo `m2` posto nella superclasse, non potrà dichiarare con la clausola `throws` eccezioni nuove che non siano sottoclassi rispetto a quelle che dichiara il metodo `m2`.
10. Dalla versione 1.4 di Java è possibile includere in un’eccezione un’altra eccezione.

Esercizio 8.b) Gestione delle asserzioni, Vero o Falso:

1. Se in un’applicazione un’asserzione non viene verificata, si deve parlare di bug.
2. Un’asserzione che non viene verificata provoca il lancio da parte della JVM di un `AssertionError`.
3. Le precondizioni servono per testare la correttezza dei parametri di metodi pubblici.
4. È sconsigliato l’utilizzo di asserzioni laddove si vuole testare la correttezza di dati inseriti da un utente.
5. Una postcondizione serve per verificare che al termine di un metodo sia verificata un’asserzione.
6. Un’invariante interna permette di testare la correttezza dei flussi all’interno dei metodi.
7. Un’invariante di classe è una particolare invariante interna che deve essere verificata per tutte le istanze di una certa classe, in ogni momento del loro ciclo di vita, tranne che durante l’esecuzione di alcuni metodi.
8. Un’invariante sul flusso di esecuzione è solitamente un’asserzione con una sintassi del tipo:

```
assert false;
```
9. Non è in alcun modo possibile compilare un programma che fa uso di asserzioni con il jdk 1.3.
10. Non è in alcun modo possibile eseguire un programma che fa uso di asserzioni con il jdk 1.3.

Soluzioni esercizi modulo 8

Esercizio 8.a) Gestione delle eccezioni e degli errori, Vero o Falso:

1. **Vero**, perché `ArithmetricException` è sottoclasse di `RuntimeException`.
2. **Falso**.
3. **Falso**, produrrà il seguente output:

```
Divisione per zero...
Finally!
```

4. **Falso**, produrrà un errore in compilazione.
5. **Falso**.
6. **Falso**, solo le sottoclassi di `Throwable`.
7. **Vero**.
8. **Falso**.
9. **Vero**.
10. **Vero**.

Esercizio 8.b) Gestione delle asserzioni, Vero o Falso:

1. **Vero**.
2. **Vero**.
3. **Falso**.
4. **Vero**.
5. **Vero**.
6. **Vero**.
7. **Vero**.
8. **Vero**.
9. **Vero**.
10. **Vero**.

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere le differenze tra eccezioni, errori ed asserzioni (unità 8.1)	<input type="checkbox"/>	
Saper gestire le varie tipologie di eccezioni con i blocchi try–catch (unità 8.2, 8.3)	<input type="checkbox"/>	
Saper creare tipi di eccezioni personalizzate e gestire il meccanismo di propagazione con le parole chiave <code>throw</code> e <code>throws</code> (unità 8.2, 8.5)	<input type="checkbox"/>	
Saper utilizzare il costrutto try with resources (unità 8.4)	<input type="checkbox"/>	
Capire e saper utilizzare le asserzioni (unità 8.6)	<input type="checkbox"/>	

Note:

Parte III

Caratteristiche avanzate

La parte III presenterà al lettore tutte le caratteristiche avanzate del linguaggio. L'approfondimento di alcuni concetti è molto elevato, e in generale i concetti stessi sono mediamente più complicati di quelli trattati nelle parti precedenti. Anche argomenti apparentemente semplici come le enumerazioni, in realtà presentano situazioni piuttosto complesse. Nei prossimi moduli esploreremo nuovi tipi di programmazione, come quella generica, quella concorrente (multithreaded) e per certi versi quella funzionale (grazie all'introduzione delle espressioni lambda). Inoltre affronteremo diverse novità di Java 8 come la nuova libreria "Date and Time API", che rivoluzionerà completamente il nostro modo di interagire con date e orari in Java, le già citate espressioni lambda che renderanno il nostro codice molto più compatto grazie alla loro sintassi minimale, la libreria delle "Stream API" che ci permetterà di usare le collezioni nella maniera più semplice ed efficiente possibile e così via. Affronteremo tutte le librerie fondamentali, il framework "Collections", l'input-output e l'interazione con i database. Dulcis in fundo il libro termina con un'introduzione al nuovo standard Java per le interfacce grafiche: JavaFX.

Enumerazioni e tipi innestati

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Comprendere cosa sono gli tipi innestati e i vantaggi che comportano (unità 9.1).
- ✓ Saper elencare le proprietà fondamentali delle classi innestate (unità 9.1).
- ✓ Saper definire ed utilizzare le classi anonime (unità 9.2).
- ✓ Comprendere l'utilizzo e l'utilità delle enumerazioni (unità 9.3).
- ✓ Comprendere l'utilizzo e l'utilità delle caratteristiche avanzate delle enumerazioni (unità 9.3).

Quando si cerca di spiegare un linguaggio di programmazione ai neofiti, uno dei problemi più grandi è quello di introdurre i concetti in maniera graduale. Ci sono alcuni concetti importanti di cui non abbiamo ancora iniziato a parlare volutamente. Questo per non confondere le idee a chi inizia e per evitare che questi concetti non vengano compresi appieno. Per esempio se avessimo iniziato a parlare dal primo modulo delle interfacce, non tutti avrebbero potuto apprezzarne la definizione. In questo libro è stato quindi deciso di proporre un approccio graduale, ecco perché spesso abbiamo trovato rimandi a moduli successivi. Nel modulo 2 abbiamo introdotto i componenti fondamentali della programmazione Java, ma ci siamo limitati davvero all'essenziale. Tanti altri concetti sono fondamentali per la programmazione Java, e tanti altri sono comunque importanti per poter comprendere appieno il linguaggio. È ora arrivato il momento quindi di introdurre le enumerazioni, che al pari delle classi e delle interfacce (e come vedremo delle annotazioni) vengono solitamente salvate in file con suffisso `.java`. Benché non sia complicato il concetto, se le avessimo introdotte prima non avremmo potuto parlare anche delle loro caratteristiche avanzate. Prima di introdurre le enumerazioni parleremo anche delle classi innestate. Nel titolo di questo modulo abbiamo parlato di **tipi innestati** perché lo stesso discorso che faremo per le classi si potrà generalizzare anche per le interfacce, le enumerazioni e le annotazioni.

9.1 Classi innestate: classi interne

Nel modulo 2 abbiamo introdotto i principali componenti di un'applicazione Java. Ci sono alcuni concetti che volutamente non sono stati introdotti in quel modulo perché avrebbero creato più confusione che altro. Stiamo parlando delle classi astratte e delle interfacce che abbiamo introdotto nel modulo 6, ma anche delle classi innestate, delle classi anonime, delle enumerazioni e infine delle annotazioni. Di seguito introduciamo le classi innestate (in inglese “nested classes”).

9.1.1 Classe innestata: definizione

Una **classe innestata** (o anche classe interna nel caso non sia dichiarata statica) non è altro che una classe definita all'interno di un'altra classe. Per esempio:

```
public class Outer {  
    private String messaggio = "Nella classe ";  
    private void stampaMessaggio() {  
        System.out.println(messaggio + "Esterna");  
    }  
    /* la classe interna accede ai membri privati  
    della classe che la contiene */  
    public class Inner {  
        public void metodo() {  
            System.out.println(messaggio + "Interna");  
        }  
        public void chiamaMetodo() {  
            stampaMessaggio();  
        }  
        //...  
    }  
    //...  
}
```

Il vantaggio di implementare una classe all'interno di un'altra riguarda principalmente il risparmio di codice. Infatti la classe interna ha accesso ai membri della classe esterna anche se dichiarati `private`. Nell'esempio, il metodo `metodo()` della classe interna `Inner` utilizza la variabile privata d'istanza `messaggio`, definita però nella classe esterna `Outer`. Stesso discorso per il metodo `chiamaMetodo()` della classe interna che chiama il metodo della classe esterna `stampaMessaggio()`. Dal punto di vista object oriented invece, di solito nessun vincolo o requisito dovrebbe consigliarci l'implementazione di una classe innestata. Questo significa che è sempre possibile evitarne l'utilizzo. Esistono dei casi dove però tali costrutti sono effettivamente molto comodi. Per esempio nella creazione di classi per la gestione degli eventi sulle interfacce grafiche, ma tale argomento sarà affrontato in dettaglio nell'apposita appendice Q dedicata alle interfacce grafiche.

Eppure anche per le classi innestate esiste la possibilità di utilizzare i modificatori d'accesso come per i membri di una classe. Quindi, per le classi ordinarie è possibile solo usare `public` oppure non usare modificatori, ed abbiamo visto nel modulo 5 come questo implichi che quella classe sia accessibile al di fuori del package oppure no. Invece per le classi innestate è possibile anche utilizzare `protected` e `private`, con le stesse regole che esistono per gli attributi di una classe, anche se non si tratta di attributi.

Inoltre è possibile anche utilizzare altri modificatori come `static`. Nel prossimo paragrafo vengono presentate le regole che governano le classi interne: sono complesse e difficili da ricordare.

9.1.2 Classi innestate: proprietà

Le seguenti proprietà vengono riportate per completezza. Tuttavia non ci dilungheremo sull'effettivo impiego di queste regole, in quanto le classi innestate statiche sono un costrutto utilizzato abbastanza

raramente. Sicuramente però, il programmatore Java esperto può sfruttare con profitto tali proprietà per risolvere problemi di non facile soluzione.

Una classe innestata:

1. deve avere un identificatore differente dalla classe che la contiene.
2. Si possono utilizzare tutti i modificatori d'accesso per dichiarare una classe innestata. Questo implica che gli oggetti di una classe innestata sono istanziabili solo negli scope definiti dai modificatori come descritto nel paragrafo 5.8.
3. Per istanziare oggetti di una classe innestata (non privata) al di fuori della classe in cui è stata dichiarata, bisogna utilizzare una sintassi speciale:

NomeClasseEsterna.nomeClasseInterna. In particolare, se volessimo istanziare una classe interna al di fuori della classe in cui è definita, bisogna eseguire i seguenti passi (facendo riferimento all'esempio precedente):

1. istanziare la classe esterna (in cui è dichiarata la classe interna):

```
Outer outer = new Outer();
```

2. dichiarare l'oggetto che si vuole istanziare dalla classe interna tramite la classe esterna (sfruttando l'operatore dot):

```
Outer.Inner inner;
```

3. istanziare l'oggetto che si vuole istanziare dalla classe interna tramite l'oggetto istanziato dalla classe esterna (notare la chiamata all'operatore new invocato tramite l'operatore dot):

```
inner = outer.new Inner();
```

4. Ha accesso sia alle variabili d'istanza sia a quelle statiche della classe in cui è dichiarata.

5. Una classe innestata si può dichiarare anche all'interno di un metodo (o in generale in un blocco di codice come quello di un if o di un for) e in questo caso viene detta classe locale (in inglese "local class"). In questo caso le variabili locali saranno accessibili solo se dichiarate final oppure effettivamente non modificate (in inglese "effectively final"). Per esempio il seguente codice:

```
public class Outer {  
    private String stringaOuter = "JAVA";  
    public void metodoOuter() {  
        int intMetodo = 7;  
        class Inner{  
            public void metodoInner(){  
                System.out.println(stringaOuter + " " + (++intMetodo));  
            }  
        }  
    }  
}
```

produrrà un errore in compilazione perché l'espressione `++intMetodo`, prova ad assegnare un

nuovo valore ad una variabile non dichiarata nella classe interna. Sino a Java 7 era obbligatorio comunque dichiarare la variabile `final`. La regola è stata resa più permissiva per consentire un'introduzione più semplice delle espressioni lambda (cfr. modulo 15).

Si noti inoltre che non è possibile dichiarare la classe `Inner` con un modificatore come `private` o `public`. Trovandosi infatti all'interno di un metodo tale dichiarazione non avrebbe senso.

6. Se viene dichiarata statica diventa automaticamente una “top-level class”. In pratica non sarà più definibile come classe interna e non godrà della proprietà di poter accedere alle variabili d'istanza della classe in cui è definita. Infatti si definisce classe interna una classe innestata non statica.
7. Solo se dichiarata statica può dichiarare membri statici.
8. Può essere dichiarata astratta.
9. Nei metodi di una classe interna è possibile utilizzare il reference `this`. Con esso ci si può riferire ai membri della classe interna e non della classe che la contiene. Per referenziare un membro della classe esterna bisognerebbe utilizzare la seguente sintassi:

```
NomeClasseEsterna.this.nomeMembroDaReferenziare
```

Questo è in realtà necessario solo nel caso ci sia possibilità d'ambiguità tra i membri della classe interna e i membri della classe esterna. Infatti, è possibile avere una classe interna ed una esterna che dichiarano un membro con lo stesso identificatore (supponiamo una variabile d'istanza `pippo`). All'interno dei metodi della classe interna, se non specifichiamo un reference per la variabile `pippo`, è scontato che sarà considerato implicitamente il reference `this`. Quindi verrà referenziata la variabile d'istanza della classe interna. Per referenziare la variabile della classe esterna bisogna utilizzare la sintassi:

```
NomeClasseEsterna.this.nomeMembroDaReferenziare
```

Per esempio, l'output del seguente codice:

```
public class Outer2 {  
    private String stringa = "esterna";  
    public class Inner2 {  
        private String stringa = "interna";  
        public void metodoInner() {  
            System.out.println(Outer2.this.stringa + " " + this.stringa);  
        }  
    }  
    public static void main(String [] args) {  
        Outer2 outer = new Outer2();  
        Outer2.Inner2 inner = outer.new Inner2();  
        inner.metodoInner();  
    }  
}
```

sarà il seguente:

10. Infine notiamo che se compiliamo una classe che contiene una classe interna, saranno creati due file: `NomeClasseEsterna.class` e `NomeClasseEsterna$NomeClasseInterna.class`.

Esiste anche un altro tipo di classe innestata detta **classe anonima**, che sarà argomento del paragrafo 9.2.

Le precedenti regole sono applicate anche ad altre strutture dati non ancora affrontate, come le interfacce, le enumerazioni e le annotazioni. Non solo sarà possibile creare in una classe classi interne, interfacce interne, enumerazioni interne e annotazioni interne, ma anche in un'interfaccia, in unenumerazione o in un'annotazione. Tuttavia stiamo parlando di pratiche di programmazione per ora trascurabili.

9.1.3 Quando usare le classi innestate

In generale si può sempre fare a meno delle classi innestate dal punto di vista della progettazione. Il vantaggio consiste soprattutto nel risparmio di codice in determinate situazioni (come per esempio quando dobbiamo creare un gestore di eventi per un'interfaccia grafica come spiegato nell'appendice Q dedicata alle GUI). Potrebbe essere desiderabile creare una classe innestata solo in casi in cui ci sia una forte relazione esclusiva di interoperabilità tra due classi, tale da creare una dipendenza biunivoca tra le due. Per esempio una classe `Auto`, potrebbe avere bisogno di una classe `Meccanico` che effettui delle riparazioni. E un meccanico dovrà avere accesso alla macchina per poterla aggiustare (dipendenza biunivoca). Supponendo di calarci in un ben determinato contesto dove un meccanico abbia sempre un auto di riferimento, allora potrebbe apparire lecito dichiarare la classe `Meccanico` come classe interna. Infatti la soluzione senza classe innestata richiederebbe la codifica delle seguenti classi:

```
public class AutoNoInner {  
    private String statoMotore;  
    private MeccanicoNoInner meccanico;  
  
    public AutoNoInner () {  
        meccanico = new MeccanicoNoInner(this);  
    }  
  
    public void setStatoMotore(String statoMotore) {  
        this.statoMotore = statoMotore;  
    }  
  
    public String getStatoMotore() {  
        return statoMotore;  
    }  
}
```

```
public class MeccanicoNoInner {  
    private AutoNoInner auto;  
  
    public MeccanicoNoInner (AutoNoInner auto) {  
        this.auto = auto;  
    }  
  
    public void aggiustaMotore() {  
        auto.setStatoMotore ("buono");  
    }  
}
```

Invece creando `Meccanico` come classe interna, la situazione si semplifica molto:

```
public class Auto {  
    private String statoMotore;  
  
    public class Meccanico {  
        public void aggiustaMotore(){  
            statoMotore = "buono";  
        }  
    }  
}
```

Il lato negativo di quest'approccio, è che se la classe `Meccanico` dovrà poi essere usata anche fuori dalla classe `Auto`, bisognerà sempre istanziare un'auto per istanziare un meccanico come spiegato nel paragrafo precedente, con la seguente sintassi:

```
Auto auto = new Auto();  
Auto.Meccanico meccanico = auto.new Meccanico();
```

9.2 Classi anonime: definizione

Come le classi innestate le classi anonime sono state introdotte successivamente alla nascita del linguaggio: nella versione 1.2. Le classi anonime non sono altro che classi innestate, ma senza nome. Essendo classi innestate, godono delle stesse proprietà (cfr. paragrafo 5.10.2) e sono utilizzate per gli stessi scopi (soprattutto per la gestione degli eventi sulle interfacce grafiche). A differenza delle classi interne però, la dichiarazione di una classe anonima richiede anche che:

1. contestualmente alla dichiarazione della classe venga anche istanziato un suo oggetto.
2. L'esistenza di una sua superclasse o di una sua superinterfaccia di cui sfrutterà il costruttore (virtualmente nel caso di un'interfaccia). Se una classe non ha nome, non può avere un costruttore.

La sintassi a prima vista può disorientare:

```
public class Outer4 {  
    private String messaggio = "Nella classe ";
```

```

//Definizione della classe anonima e sua istanza
ClasseEsistente ce = new ClasseEsistente() {
@Override
public void metodo() {
System.out.println(messaggio+"anonima");
}
}; //Si noti il ";"
//. .
}

//Superclasse della classe anonima
public class ClasseEsistente {
public void metodo() {
System.out.println("Nella classe esistente");
}
}

```

In pratica quando si dichiara una classe anonima la si deve anche istanziare, ma come istanziare una classe senza nome? Una classe anonima deve quindi estendere sicuramente un'altra classe (nell'esempio estendeva `ClasseEsistente`) e sfruttarne un reference (grazie al polimorfismo per dati) e un costruttore (per definizione di classe anonima). Infatti la classe anonima dell'esempio usa il reference `ce` di tipo `ClasseEsistente` per referenziare un oggetto della sua sottoclasse anonima, e implicitamente come tutte le sottoclassi chiama sempre il costruttore della superclasse.

Se compiliamo una classe che contiene una classe anonima, verranno creati due file: `NomeClasseEsterna.class` e `NomeClasseEsterna$1.class`. Se introduciamo una seconda classe anonima sarà creato anche il file `NomeClasseEsterna$2.class` e così via.

Si noti che una classe anonima viene sempre dichiarata con lo scopo di fare override di uno o più metodi della classe che estende. Se infatti nell'esempio avessimo definito un nuovo metodo (non ereditato) nella classe anonima, non saremmo stati in grado di invocarlo senza un reference del tipo della classe anonima (che ovviamente non può esistere). Per esempio se nell'esempio precedente avessimo definito la classe anonima in questo modo:

```

//Definizione della classe anonima e sua istanza
ClasseEsistente ce = new ClasseEsistente() {
@Override
public void metodo() {
System.out.println(messaggio+"anonima");
}

public void metodoNonInvocabile!() {
System.out.println("Per una classe anonima non esistono "
+ "reference: impossibile chiamare questo metodo!");
}
};

//. .

```

Per quanto detto nel modulo 7 riguardante il polimorfismo, essendo la precedente una classe senza nome, non possiamo ottenere un reference della classe stessa, ma solo usarne uno della classe che estende.

Infine come le classi interne, anche le classi anonime si possono dichiarare all'interno di metodi, ed anche in questo caso c'è il vincolo di poter accedere alle variabili locali e ai parametri, solo se dichiarati `final`, o se sono effettivamente non modificabili (effectively final). Supponendo che sia stata definita la classe `Outer4`, il seguente codice è valido:

```
public class Outer4 {  
    private String messaggio = "Nella classe ";  
    public void metodoConClasse(final int a) {  
        ClasseEsistente ce = new ClasseEsistente() {  
            @Override  
            public void metodo() {  
                System.out.println(messaggio+"anonima numero "+ a);  
            }  
        };  
        ce.metodo();  
    }  
}
```

In questo caso quindi, dichiarare `final` il parametro non è necessario affinché la classe anonima la possa utilizzare (questo vale da Java 8 in poi), ma di certo non è una cattiva abitudine. In generale comunque, dichiarare un parametro `final` anche se apparentemente inutile, potrebbe rinforzare l'idea che esso non si debba cambiare. Nel caso il parametro sia un reference inoltre, dichiarandolo `final` ci metteremmo al riparo già in fase di compilazione da eventuali problemi di assegnazioni inutili (che probabilmente porterebbero a un bug). Infatti come già visto in precedenza, l'assegnazione ad un reference di un nuovo indirizzo non implica che il reference uscito dal metodo abbia davvero cambiato indirizzo (cfr. paragrafo 3.3.1 sul passaggio di parametri per valore). Ma è anche vero che quando abbiamo a che far con reference, anche se non possiamo cambiare il loro indirizzamento, è sempre possibile cambiare il loro stato interno tramite un metodo “mutator”.

Abbiamo affermato precedentemente che le regole per le classi innestate valgono anche per le interfacce, le enumerazioni e le annotazioni. Per quanto riguarda le classi anonime, è invece possibile solo estendere classi o interfacce, ma non enumerazioni o annotazioni. Se consideriamo per esempio l'interfaccia `Volante`, introdotta nel modulo 6, aggiungendoci il nuovo metodo `plana()`:

```
public interface Volante {  
    void plana();  
    void decolla();  
    void atterra();  
}
```

allora sarà possibile crearne istanze non identificate velocemente, mediante la seguente sintassi:

```
public class TestVolanteAnonymous {
```

```
public static void main(String args[]) {  
    Volante ufo = new Volante() {  
        @Override  
        public void decolla() {  
            System.out.println("Un oggetto non identificato sta " + "decolando");  
        }  
        @Override  
        public void plana() {  
            System.out.println("Un oggetto non identificato sta " + "planando");  
        }  
        @Override  
        public void atterra() {  
            System.out.println("Un oggetto non identificato " + "atterra...");  
        }  
    };  
    //Usiamo l'oggetto della classe anonima.  
    ufo.decolla();  
    ufo.plana();  
    ufo.atterra();  
}
```

Si noti come sembra di istanziare oggetti direttamente dall'interfaccia, che viene implementata contestualmente all'istanza dell'oggetto. Si noti anche che la sintassi sembra faccia utilizzare il costruttore di un interfaccia! Ma sappiamo che le interfacce non hanno costruttori, quindi con la sintassi:

```
new Volante() { . . . }
```

stiamo usando un costruttore virtuale. È possibile anche estendere classi astratte per ridefinire i metodi astratti.

9.2.1 Quando usare le classi anonime

Un utilizzo particolarmente utile delle classi anonime è quando vengono dichiarate al momento di passare un parametro in un metodo. Probabilmente questo è il modo più utile per utilizzarle. Stiamo parlando di creare delle specializzazioni con override create al volo. Per esempio consideriamo la seguente classe `Pilota` che dichiara un metodo che prende in input un oggetto `Volante`:

```
public class Pilota {  
    private String nome;  
  
    public Pilota (String nome) {  
        this.nome = nome;  
    }  
  
    public void fattiUnGiro(Volante volante) {  
        volante.decolla();  
    }  
}
```

```

volante.plana();
volante.atterra();
}

public void setNome(String nome) {
this.nome = nome;
}

public String getNome() {
return nome;
}
}

```

Possiamo usare le classi anonime in maniera dinamica come nel seguente esempio:

```

public class TestPilota {
public static void main(String args[]) {
Pilota pilota = new Pilota("Simone");
pilota.fattiUnGiro(new Volante() {
@Override
    public void decolla() {
System.out.println("Un oggetto non identificato sta " + "decollando");
}
@Override
public void plana() {
System.out.println("Un oggetto non identificato sta " + "planando");
}
@Override
public void atterra() {
System.out.println("Un oggetto non identificato sta " + "atterrando");
}
} );
}
}

```

In questo caso possiamo cambiare rapidamente l'implementazione dei metodi, senza aver magari mai dichiarato la classe, e senza neanche mai aver dato un reference all'oggetto creato. È un modo di lavorare meno object oriented ma molto utile. In pratica possiamo dire di aver passato ad un metodo del codice da eseguire! Ma questo è davvero molto potente, cerchiamo di capire il perché.

Uno dei così detti “anti-pattern” della programmazione (il contrario dei pattern, ovvero qualcosa da evitare) è il cosiddetto “copia-incolla”. Certo è comodo evitare di scrivere due volte lo stesso pezzo di codice, ma se lo stiamo facendo, dobbiamo renderci conto che stiamo sbagliando a priori: dovremmo addirittura sentirci in colpa! Questo perché in effetti stiamo aprendo la strada a potenziali bachi. Infatti se un giorno modificheremo uno dei due pezzi di codice perché ci accorgiamo di un bug, è probabile che non lo faremo nell'altro che rimarrà bacato. Dopotutto uno dei paradigmi fondamentali dell'Object Orientation e della buona programmazione in generale è il riuso. Il nostro scopo deve essere quindi quello di evitare a tutti i costi il copia-incolla.

Per esempio se ci troviamo a dover riscrivere un pezzo di codice in due classi diverse, è possibile che queste due classi possano trovarsi in qualche relazione di ereditarietà. Se avessimo la classe `Pilota` e `Persona` che dichiarano le proprietà (attributi privati con metodi accessor e mutator) `nome`, `cognome` e `codiceFiscale`, è facile pensare che `Pilota` possa estendere la classe `Persona`.

Invece se due classi condividono in due metodi diversi un frammento di codice identico è molto probabile che quel pezzo di codice si possa astrarre in un metodo da invocare, magari spostato in una terza classe di utilità.

Se invece il pezzo di codice in comune è in due metodi della stessa classe, allora potrebbe essere possibile estrapolare questo pezzo di codice in un metodo privato che viene semplicemente invocato dai due metodi originari.

E se questo pezzo di codice in comune differisce di poco (magari di una sola riga) da un altro? In alcuni casi è comunque possibile astrarre in un metodo il frammento di codice, facendo dipendere il frammento di codice differente da un parametro passato al nuovo metodo. Cerchiamo di spiegarci meglio con un esempio. Per esempio consideriamo la seguente classe `Film`:

```
public class Film {  
    private String nome;  
    private String genere;  
    private int mediaRecensioni;  
  
    public Film (String nome, String genere, int mediaRecensioni) {  
        this.nome = nome;  
        this.genere = genere;  
        this.mediaRecensioni = mediaRecensioni;  
    }  
    // Metodi set e get omessi . . .  
    public String toString(){  
        return getNome();  
    }  
}
```

Andiamo a scrivere una classe `Videoteca`, che contiene due metodi praticamente uguali a parte una riga di codice:

```
public class Videoteca {  
    private Film[] films;  
  
    public Videoteca () {  
        films = new Film[10];  
        caricaFilms();  
    }  
    public void setFilms(Film[] films) {  
        this.films = films;  
    }  
  
    public Film[] getFilms() {  
        return films;  
    }
```

```

public Film[] getFilmDiFantascienza() {
    Film [] filmDiFantascienza = new Film[10];
    for (int i = 0, j= 0; i< 10;i++) {
        if ("Fantascienza".equals(films[i].getGenere())) {
            filmDiFantascienza[j] = films[i];
            j++;
        }
    }
    return filmDiFantascienza;
}

public Film[] getBeiFilm() {
    Film [] beiFilms = new Film[10];
    for (int i = 0, j= 0; i< 10;i++) {
        if (films[i].getMediaRecensioni() >3) {
            beiFilms[j] = films[i];
            j++;
        }
    }
    return beiFilms;
}

private void caricaFilms() {
    //Caricamento film...
}
}

```

I metodi `getFilmDiFantascienza()` e `getBeiFilm()` che filtrano rispettivamente film di fantascienza e i film con una media di recensione superiore a 3, sono praticamente identici a meno dei nomi delle variabili locali (che però sono solo nomi, e potremmo anche chiamare allo stesso modo) e il controllo nella clausola `if`. L'algoritmo non è nemmeno banale perché nel ciclo `for` abbiamo definito due indici che vengono incrementati in momenti diversi: `i` per scorrere l'array `films` (variabile d'istanza) e `j` per l'array da riempire e ritornare. Sembra un ostacolo insormontabile evitare il copia-incolla, ed infatti nella maggior parte dei casi il programmatore Java medio lascia il codice così, mettendo a rischio di bug il programma.

Con la seguente classe stampiamo i film di fantascienza:

```

public class TestVideoteca {
    public static void main(String args[]) {
        Videoteca videoteca = new Videoteca();
        System.out.println("Film di Fantascienza:");
        Film[] filmDiFantascienza = videoteca.getFilmDiFantascienza();
        for (Film film: filmDiFantascienza) {
            if (film != null) {
                System.out.println(film);
            }
        }
    }
}

```

```
}
```

La soluzione consiste nell'utilizzo di classi anonime. Per prima cosa dobbiamo creare una semplice interfaccia in questo modo che astrae il concetto di filtro:

```
public interface FiltroFilm {  
    boolean filtra(Film film);  
}
```

Poi eliminiamo i due metodi incriminati e li sostituiamo con il seguente:

```
public Film[] getFilmFiltrati(FiltroFilm filtroFilm) {  
    Film [] filmFiltrati = new Film[10];  
    for (int i = 0, j= 0; i< 10;i++) {  
        if (filtroFilm.filtra(films[i])) {  
            filmFiltrati[j] = films[i];  
            j++;  
        }  
    }  
    return filmFiltrati;  
}
```

A questo punto possiamo filtrare i nostri film con il seguente codice:

```
//...  
Videoteca videoteca = new Videoteca();  
System.out.println("Bei Film:");  
Film[] beiFilms = videoteca.getFilmFiltrati(new FiltroFilm() {  
    public boolean filtra(Film film) {  
        return film.getMediaRecensioni() >3;  
    }  
});  
//...  
System.out.println("\nFilm di Fantascienza:");  
Film[] filmDiFantascienza = videoteca.getFilmFiltrati(new FiltroFilm() {  
    public boolean filtra(Film film) {  
        return "Fantascienza".equals(film.getGenere());  
    }  
});  
//...
```

In questo modo abbiamo passato il codice da eseguire al metodo, evitando il copia-incolla! Possiamo anche inventare nuovi filtri definendoli al volo.

Questa soluzione è molto avanzata e flessibile, forse è la più complessa che abbiamo visto sino ad ora, tanto che molti programmatore Java non l'hanno mai usata! Questo tipo di soluzione è anche molto usata per la gestione delle azioni sulle interfacce grafiche come è possibile leggere nell'appendice Q dedicata.

Il lato negativo di quest'approccio alla programmazione è che non è favorita la leggibilità e

bisogna scrivere molto codice. Ma con l'introduzione delle espressioni lambda in Java 8, vedremo nel modulo 15 che sarà possibile ottenere lo stesso risultato con un minore sforzo e una sintassi molto più compatta e potente.

9.3 Tipi Enumerazioni

Gli enumerated types, che traduciamo come tipi enumerazioni, o più semplicemente solo con enumerazioni o enum, sono uno dei componenti fondamentali della programmazione Java che non abbiamo ancora introdotto. Si tratta di una nuova tipologia di struttura dati, che si aggiunge alle classi, alle interfacce ed alle annotazioni (che approfondiremo in un modulo dedicato), e quindi si possono salvare in file con suffisso .java. Questa volta invece di utilizzare `class` o `interface`, useremo la parola chiave `enum`. Le enumerazioni sono strutture dati somiglianti alle classi, ma con proprietà particolari. Facciamo subito un esempio:

```
public enum MiaEnumerazione {  
    UNO, DUE, TRE;  
}
```

Abbiamo appena definito un'enum di nome `MiaEnumerazione`, definendo tre suoi elementi (detti anche valori) che si chiamano `UNO`, `DUE`, `TRE`.

In pratica è come se avessimo definito un nuovo tipo. Infatti, un'enumerazione viene trasformata dal compilatore in una classe che estende la classe astratta `Enum` (package `java.lang`). Gli elementi di questa enum sono implicitamente di tipo `MiaEnumerazione` e quindi non va specificato il tipo. Essi vengono semplicemente definiti separandoli con virgolette. Si tratta di costanti statiche, ma non bisogna dichiararle né `final` né `public` né `static` in quanto in un'enum lo sono già implicitamente. Si noti che nell'esempio dopo la lista dei valori il ";" non è necessario, ma consigliato. Diventerà infatti necessario nel caso vengano definiti altri elementi nell'enumerazione, come per esempio metodi.

Ogni elemento di `MiaEnumerazione` è di tipo `MiaEnumerazione`. È questa la caratteristica delle enum più difficile da digerire all'inizio. In sostanza, definita un'enumerazione, si definiscono anche tutte le sue possibili istanze. Non si possono istanziare altre `MiaEnumerazione` oltre a quelle definite da `MiaEnumerazione` stessa. Ecco perché sembra essere più corretto definire i suoi elementi come "valori" dell'enumerazione. Trattandosi di costanti, la convenzione Java (cfr. modulo 3) consiglia di definire gli elementi di un'enumerazione con caratteri maiuscoli.

Ricordiamo che come separatore di parole può essere utilizzato il carattere underscore ("_"). Inoltre le enumerazioni vanno definite con la stessa convenzione delle classi. La sintassi di un'enum non si limita solo a quanto appena visto. Esistono tante altre caratteristiche che può avere un'enumerazione, come implementare interfacce, definire costruttori, metodi etc.

9.3.1 Ereditarietà e polimorfismo con enum

Un'enum non si può estendere né può estendere un'altra enum o un'altra classe. Infatti le enumerazioni sono trasformate dal compilatore in classi che estendono la classe `java.lang.Enum`.

Questo significa che erediteremo dalla classe `java.lang.Enum` diversi metodi che possiamo tranquillamente invocare sugli elementi dell'enum. Per esempio il metodo `toString()` è definito in modo tale da restituire il nome dell'elemento, per cui:

```
System.out.println(MiaEnum.Enumeration.UNO);
```

produrrà il seguente output:

```
UNO
```

È anche possibile fare override dei metodi di `java.lang.Enum`, compreso il metodo `toString()`.

`Enum` dichiara anche un metodo complementare a `toString()`: il metodo statico `valueOf()`. Per esempio:

```
System.out.println(MiaEnum.Enumeration.valueOf("UNO"));
```

stamperà:

```
MiaEnum.Enumeration.UNO
```

Inoltre `Enum` definisce il metodo `final ordinal()`. Tale metodo ritorna la posizione all'interno dell'enum di un suo elemento. Come al solito l'indice parte da zero e questo metodo è sfruttabile all'interno di cicli.

Le enumerazioni definiscono anche il metodo `values()`. Questo metodo consente di iterare sui valori di un'enumerazione. Tale pratica potrebbe servire quando non conosciamo l'enumerazione in questione. Per esempio, sfruttando un ciclo `for` migliorato, possiamo stampare i contenuti dell'enumerazione `MiaEnum`:

```
for (MiaEnum miaEnum : MiaEnum.values()) {  
    System.out.println(miaEnum);  
}
```

Come abbiamo già asserito, le enumerazioni sono trasformate dal compilatore in classi che estendono la classe `java.lang.Enum`. In particolare, ogni enum estende implicitamente la classe astratta `java.lang.Enum` (che non è un'enumerazione). Ecco perché possono usufruire o sottoporre ad override i metodi di `Enum`. Attenzione che il compilatore non permetterà allo sviluppatore di creare classi che estendono direttamente la classe `Enum`. Essa è una classe speciale creata appositamente per supportare il concetto di enumerazione.

Un'enumerazione non può estendere altre enumerazioni né altre classi. Infatti, se dovesse estendere un'altra enumerazione o un'altra classe, il compilatore non potrebbe fare estendere ad essa la classe `java.lang.Enum` per le regole dell'ereditarietà singola che si applica alle classi.

È invece possibile far implementare un'interfaccia ad un'enum. Infatti possiamo dichiarare in un'enum tutti i metodi che vogliamo, compresi quelli da implementare che fanno parte di un'interfaccia. Questo significa che se vogliamo sfruttare il polimorfismo per dati, possiamo farlo

solo tramite l'utilizzo di interfacce o un reference di tipo `enum`. Per esempio consideriamo la seguente interfaccia:

```
public interface Numeratore {  
    void stampaIndice();  
}
```

riscriviamo l'enumerazione facendogli implementare l'interfaccia precedente:

```
public enum MiaEnumerazione2 implements Numeratore {  
    UNO, DUE, TRE;  
    @Override  
    public void stampaIndice() {  
        System.out.println("Indice: " + this.ordinal());  
    }  
}
```

e sarà possibile usare il polimorfismo per dati sfruttando l'interfaccia `Numeratore` nel seguente modo:

```
Numeratore n = MiaEnumerazione2.DUE;  
n.stampaIndice();
```

che darà come output:

```
Indice: 1
```

In realtà è possibile anche usare un reference di tipo `java.lang.Enum` per utilizzare il polimorfismo per dati:

```
Enum e = MiaEnumerazione2.UNO;
```

ma con un reference di tipo `java.lang.Enum` non potremo chiamare il metodo `stampaIndice()` come visto precedentemente, per le regole del polimorfismo per dati.

L'ultima istruzione in realtà verrà sì compilata, ma verrà segnalato un warning dal compilatore. Infatti non è stata utilizzata correttamente la classe `Enum` che in realtà è definita usando un “tipo generico”. Spiegheremo dettagliatamente questo argomento più avanti.

Non potendo estendere un'enum non la si potrà dichiarare `abstract`. Quindi, nel momento in cui implementiamo un'interfaccia in un'enum, dovremo implementare obbligatoriamente tutti i metodi ereditati.

9.3.2 Metodi, variabili, costruttori e tipi innestati in un'enumerazione

In un'enum è possibile anche creare variabili, metodi e costruttori. Questi ultimi però sono implicitamente dichiarati `private` e non è possibile utilizzarli se non nell'ambito dell'enumerazione stessa. Per esempio, con il seguente codice viene ridefinita l'enum `MiaEnumerazione` per la terza volta:

```
public enum MiaEnumerazione3 {  
    ZERO(), UNO(1), DUE(2), TRE(3);  
    private int valore;  
    private MiaEnumerazione3() {  
    }  
    MiaEnumerazione3(int valore) {  
        setValore(valore);  
    }  
    public void setValore(int valore){  
        this.valore = valore;  
    }  
    public int getValore(){  
        return this.valore;  
    }  
    @Override  
    public String toString(){  
        return "" + valore;  
    }  
}
```

Si noti come il costruttore sia sfruttato quando vengono definiti gli elementi dell'enum. Per le enumerazioni valgono le seguenti regole:

- 1. Qualsiasi dichiarazione deve seguire la dichiarazione degli elementi dell'enumerazione.** Se anteponessimo una qualsiasi delle dichiarazioni aggiunte alla lista degli elementi, otterremmo un errore in compilazione. In questo caso la dichiarazione degli elementi deve terminare esplicitamente con un ";" ed è buona abitudine che sia sempre così.
- 2. È possibile dichiarare un qualsiasi numero di costruttori (sfruttando l'overload) che implicitamente saranno considerati `private`.** Nell'esempio abbiamo due costruttori, di cui uno abbiamo esplicitato (ma è ridondante) il modificatore `private`. Se avessimo dichiarato esplicitamente un costruttore `public` avremmo ottenuto un errore in compilazione. Per il resto valgono le regole applicate ai costruttori delle classi. Come per le classi anche per le enumerazioni se non inseriamo costruttori il compilatore ne aggiungerà uno per noi senza parametri (il costruttore di default). Sempre come per le classi, il costruttore di default non verrà inserito nel momento in cui ne inseriamo noi uno esplicitamente, come nell'esempio precedente.
- 3. Quando sono esplicitati i costruttori come in questo caso, i valori dell'enum possono**

utilizzarli. Basta osservare il codice dell'esempio. I valori UNO, DUE e TRE utilizzano il costruttore che prende in input un intero. Il valore ZERO, invece, utilizza il costruttore senza parametri. In particolare è possibile notare come il valore ZERO sia dichiarato in maniera diversa rispetto agli altri valori. Infatti, esplicita due parentesi vuote, che sottolineano come stia utilizzando il costruttore senza parametri. Tale sintassi è assolutamente superflua, ed è stata riportata solo per preparare il lettore a strane sorprese. Se nell'esempio avessimo avuto un unico costruttore (quello che prende come parametro in input un intero) tutti gli elementi dell'enum avrebbero obbligatoriamente dovuto utilizzare quell'unico costruttore.

- 4. Nelle enumerazioni è possibile dichiarare tipi innestati (classi, interfacce, enumerazioni e annotazioni) esattamente come lo si può fare nelle classi, nelle interfacce (e nelle annotazioni).** Non bisogna fare alcuna attenzione particolare a queste dichiarazioni, se non come già asserito nel dichiararli dopo i valori dell'enum. Per esempio, è possibile anche dichiarare un enum in un enum, come segue:

```
public enum MyEnum {  
    ENUM1 (), ENUM2;  
    public enum MyEnum2 {a,b,c}  
}
```

9.3.3 Quando utilizzare un'enum

Non si può dire che le enum consentano di creare codice che non si possa creare con le classi, semmai si può affermare che permetteranno di creare codice più robusto in alcuni casi. È consigliato l'uso dell'enum ogni qualvolta ci sia bisogno di dichiarare un numero finito di valori da utilizzare per gestire il flusso di un'applicazione. Facciamo un esempio. Prima dell'avvento delle enumerazioni in Java spesso si rendeva necessario creare costanti simboliche per rappresentare valori. Tali costanti venivano spesso definite di tipo intero oppure stringa. A volte si raccoglievano le costanti all'interno di interfacce dedicate. Per esempio:

```
public interface Azione {  
    public static final int AVANTI = 0;  
    public static final int INDIETRO = 1;  
    public static final int FERMO = 2;  
}
```

Oltre al fatto che le interfacce si potrebbero usare per scopi più object oriented, il motivo fondamentale per cui questo approccio è considerato sconsigliabile, è che a livello di compilazione non si possono esplicitare vincoli che impediscano all'utente di utilizzare scorrettamente un tipo.

Consideriamo per esempio il seguente codice:

```
public void esegui(int azione) {  
    switch (azione) {  
        case Azione.AVANTI:  
            vaiAvanti();  
            break;
```

```
case Azione.INDIETRO:  
vaiIndietro();  
break;  
case Azione.FERMO:  
fermati();  
break;  
}  
}
```

Nessun compilatore potrebbe rilevare che la seguente istruzione non è un'istruzione valida:

```
oggetto.esegui(3);
```

È possibile aggiungere una clausola `default` al costrutto `switch`, dove si gestisce in qualche modo il problema, ma questo sarà eseguito solo in fase di runtime dell'applicazione e non in fase di compilazione.

Un'implementazione più robusta della precedente potrebbe richiedere l'utilizzo di una classe come la seguente:

```
public class Azione {  
private String nome;  
public static final Azione AVANTI = new Azione("AVANTI");  
public static final Azione INDIETRO = new Azione("INDIETRO");  
public static final Azione FERMO = new Azione("FERMO");  
public Azione(String nome){  
setNome(nome);  
}  
public void setNome(String nome) {  
this.nome = nome;  
}  
public String getNome(){  
return this.nome;  
}  
}
```

In tale caso il metodo `esegui()` dovrebbe essere modificato in modo tale da sostituire il costrutto `switch` con un costrutto `if`:

```
public void esegui(Azione azione) {  
if (azione == Azione.AVANTI) {  
vaiAvanti();  
}  
else if (azione == Azione.INDIETRO) {  
vaiIndietro();  
}  
else if (azione == Azione.FERMO) {  
fermo();  
}
```

```
}
```

Purtroppo anche in questo caso però il problema rimane (anche se l'astrazione con una classe sembra migliore rispetto a quella con un'interfaccia). È sempre possibile passare `null` al metodo `esegui()`. Qualsiasi controllo volessimo inserire potrebbe risolvere il problema solo in fase di runtime dell'applicazione.

In casi come questo l'uso di un'enumerazione mette tutti d'accordo. Se infatti creiamo la seguente enumerazione:

```
public enum AzioneEnum {  
    AVANTI, INDIETRO, FERMO;  
}
```

senza modificare il codice del metodo `esegui()` (è possibile utilizzare un tipo `enum` anche come variabile di test di un costrutto `switch`) avremo risolto il problema.

9.3.4 Enumerazioni innestate (in classi) o enumerazioni membro

Come già asserito anche le enumerazioni si possono innestare nelle classi. Per esempio è possibile scrivere codice come il seguente:

```
public class Volume {  
    public enum Livello {ALTO, MEDIO, BASSO};  
    // implementazione della classe . . .  
}
```

Se volessimo stampare un elemento della enumerazione all'interno della classe con il metodo `toString()`, è possibile utilizzare la seguente sintassi:

```
System.out.println(Livello.ALTO);
```

nel caso in cui ci si trovasse al di fuori della classe, dovremmo utilizzare la sintassi:

```
System.out.println(Volume.Livello.ALTO);
```

è anche possibile utilizzare l'incapsulamento, ma trattandosi di costanti statiche non si corrono grossi pericoli.

Una enum innestata è statica implicitamente. Infatti il seguente codice è valido:

```
public class Volume {  
    public enum Livello {ALTO, MEDIO, BASSO};  
    // implementazione della classe . . .  
    public static void main(String args[]) {  
        System.out.println(Livello.ALTO);  
    }  
}
```

Se `Livello` non fosse statica non avremmo potuto utilizzarla direttamente in un metodo statico come `il main()`.

Su altri testi le enumerazioni innestate sono chiamate semplicemente “enumerazioni membro” (membro di una classe; cfr. modulo 2). Per non avviare una sterile discussione su come sia più corretto chiamare tale costrutto affermiamo che sono solo punti di vista differenti.

Si noti come gli import statici descritti nel paragrafo 5.9.4 si adattino benissimo ad essere utilizzati per importare i valori delle enumerazioni, o anche i tipi innestati all'interno delle enumerazioni. In entrambi i casi infatti stiamo parlando di elementi statici. Supponendo di dichiarare la classe `Volume` appartenente al package `musica`:

```
package musica;

public class Volume {
    // . . .
```

in un'altra classe potremmo importare il valore `ALTO` dell'enum `Livello` innestata all'interno della classe `Volume`, con la seguente sintassi:

```
import static musica.Volume.Livello.ALTO;

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

Si noti che verrà stampato il risultato del metodo `toString()` chiamato sull'elemento `ALTO`.

9.3.5 Enumerazioni e metodi specifici degli elementi

Ma è proprio impossibile estendere un'enum? In effetti no! È possibile infatti che una certa enumerazione sia estesa dai suoi stessi elementi sfruttando la sintassi delle classi anonime. Si possono definire metodi nell'enumerazione e fare override di essi con i suoi elementi. Consideriamo la seguente enumerazione `PuntiCardinali`:

```
public enum PuntiCardinali {
    NORD {
        @Override
        public void test() {
            System.out.println("metodo di NORD");
        }
    },
    SUD {
        @Override
        public void test() {
            System.out.println("metodo di SUD");
        }
    },
    EST {
        @Override
        public void test() {
            System.out.println("metodo di EST");
        }
    },
    OVEST {
        @Override
        public void test() {
            System.out.println("metodo di OVEST");
        }
    }
}
```

```
SUD, OVEST, EST;  
public void test() {  
    System.out.println("metodo dell'enum");  
}  
}
```

È stato definito un metodo che abbiamo chiamato `test()` e dovrebbe stampare la stringa “metodo dell’enum”. Però l’elemento `NORD`, con una sintassi simile a quella delle classi anonime, dichiara anch’esso lo stesso metodo, sottponendolo a override. Infatti il compilatore tramuterà `NORD` proprio in una classe anonima che estenderà `PuntiCardinali`. Quindi l’istruzione:

```
PuntiCardinali.NORD.test();
```

stamperà:

```
metodo di NORD
```

mentre l’istruzione:

```
PuntiCardinali.SUD.test();
```

stamperà:

```
metodo dell'enum
```

perché `SUD` non ha fatto override di `test()`.

9.3.6 Switch e enum

Nel modulo 4 abbiamo introdotto il costrutto `switch`. Ora che conosciamo le enumerazioni dobbiamo capire come è possibile utilizzare questo concetto nel costrutto. Se un costrutto `switch` definisce come variabile di test un’enumerazione, tutte le istanze di tale enumerazione possono essere possibili costanti per i case. Per esempio, tenendo presente l’enumerazione `Livello` di cui sopra, consideriamo il seguente frammento di codice:

```
switch (getLivello()) {  
    case ALTO:  
        System.out.println(Livello.ALTO);  
        break;  
    case MEDIO:  
        System.out.println(Livello.MEDIO);  
        break;  
    case BASSO:  
        System.out.println(Livello.BASSO);  
        break;  
}
```

La variabile di test è di tipo `Livello` e le costanti dei case sono gli elementi dell'enumerazione stessa. Si noti come gli elementi non abbiano bisogno di essere referenziati con il nome dell'enumerazione nel seguente modo:

```
case Livello.ALTO:  
    System.out.println(Livello.ALTO);  
    break;  
case Livello.MEDIO:  
    System.out.println(Livello.MEDIO);  
    break;  
case Livello.BASSO:  
    System.out.println(Livello.BASSO);  
    break;
```

infatti la variabile di test fornisce già la sicurezza del tipo.

Nonostante in un costrutto `switch` che si basa su un'enum sia escluso che la clausola `default` possa essere eseguita durante il runtime (questo è uno dei vantaggi delle enum rispetto ai "vecchi" approcci) è comunque buona norma utilizzarne una. Infatti è facile che l'enumerazione subisca nel tempo delle aggiunte. Questo è particolarmente vero se il codice è condiviso tra più programmatori. In tal caso ci sono due approcci da consigliare. Il primo è più soft e consiste nel gestire comunque eventuali nuovi tipi in maniera generica. Per esempio:

```
default:  
    System.out.println(getLivello());
```

Il secondo metodo è senz'altro più robusto in quanto basato sulle asserzioni. Per esempio:

```
default:  
    assert false: "valore dell'enumerazione nuovo: " + getLivello();
```

Si noti che l'asserzione dovrebbe fare emergere il problema durante i test se l'enumerazione `Livello` fosse stata ampliata con nuovi valori.

Nel caso non si voglia implementare la clausola `default`, possiamo comunque farci aiutare dal compilatore ove l'enumerazione coinvolta nello `switch` si evolva. Infatti se in uno `switch` non vengono contemplati tutti i case dell'enumerazione, come di seguito:

```
case ALTO:  
    System.out.println(Livello.ALTO);  
    break;  
case BASSO:  
    System.out.println(Livello.BASSO);  
    break;
```

allora in compilazione il compilatore ci segnalerà un warning (sempre nel caso stiamo utilizzando un IDE, oppure stiamo usando l'opzione `-Xlint` per compilare da riga di comando (cfr. paragrafo 8.5.1)).

Riepilogo

In questo modulo abbiamo visto essenzialmente due argomenti principali: le **classi innestate** e le **enumerazioni**. La conoscenza delle classi innestate è propedeutica alla definizione avanzata delle enumerazioni. Entrambi gli argomenti sono abbastanza semplici da imparare se ci si limita all'utilizzo di base. Diventano molto più complessi quando si vogliono applicare le caratteristiche avanzate di queste nuove definizioni. Particolarmente ostiche sono le proprietà delle classi innestate, poco conosciute dalla maggior parte dei programmati Java. Molto importanti dal punto di vista dell'utilizzo sono le **classi anonime**, che sino a Java 7 hanno rappresentato il modo più dinamico, sintetico e funzionale di scrivere codice Java. Con l'introduzione delle espressioni lambda, molto probabilmente le classi anonime perderanno questo primato.

La seconda parte del modulo è probabilmente più interessante. Infatti, abbiamo definito le enumerazioni con il supporto di semplici esempi. Dopo aver presentato le ragioni per cui le enumerazioni costituiscono un vero punto di forza di Java, ne abbiamo presentato le proprietà. Inoltre, abbiamo cercato di capire che lavoro svolge il compilatore per noi. Abbiamo notato come tutte le enumerazioni, inoltre, estendano la classe `java.lang.Enum` ereditandone i metodi. Dopo aver presentato anche le caratteristiche avanzate delle enumerazioni, come i metodi specifici degli elementi, ne abbiamo analizzato gli impatti sul linguaggio. Infine, abbiamo visto come un costrutto come lo `switch` e gli **import static** siano adatti ad utilizzare enumerazioni, e come queste siano spesso innestate in classi.

Esercizi modulo 9

Esercizio 9.a) Tipi innestati, Vero o Falso:

1. Una classe innestata è una classe che viene dichiarata all'interno di un'altra classe.
2. Una classe anonima è anche innestata ma non ha nome. Inoltre, per essere dichiarata, deve per forza essere istanziata
3. Le classi innestate non sono necessarie per l'Object Orientation.
4. Una classe innestata deve essere per forza istanziata.
5. Per istanziare una classe innestata pubblica a volte bisogna istanziare prima la classe esterna.
6. Una classe innestata dichiarata `private` deve dichiarare anche i metodi "set" e "get" per poter essere utilizzata da una terza classe.
7. Una classe innestata non può avere lo stesso nome della classe che la contiene.
8. Una classe anonima può avere lo stesso nome della classe che la contiene.
9. Una classe innestata può accedere a membri statici della classe che la contiene solo se è dichiarata statica.
10. Una classe innestata non può essere dichiarata astratta.

Esercizio 9.b) Enumerazioni, Vero o Falso:

- Le enumerazioni non si possono istanziare se non all'interno della definizione dell'enumerazione stessa. Infatti possono avere solamente costruttori private.
- Le enumerazioni possono dichiarare metodi e possono essere estese da classi che possono sottoporne a override i metodi. Non è però possibile che un'enum estenda un'altra enum.
- Il metodo `values()` appartiene ad ogni enumerazione ma non alla classe `java.lang.Enum`.
- Il seguente codice viene compilato senza errori:

```
public enum MyEnum {  
    public void metodo1() {  
  
    }  
    public void metodo2() {  
  
    }  
    ENUM1, ENUM2;  
}
```

- Il seguente codice viene compilato senza errori:

```
public enum MyEnum {  
    ENUM1 {  
        public void metodo() {  
  
    }  
    }, ENUM2;  
    public void metodo2() {  
  
    }  
}
```

- Il seguente codice viene compilato senza errori:

```
public enum MyEnum {  
    ENUM1 (), ENUM2;  
    private MyEnum(int i) {  
  
    }  
}
```

- Il seguente codice viene compilato senza errori:

```
public class Volume {  
    public enum Livello {  
        ALTO, MEDIO, BASSO  
    };  
    // implementazione della classe . . .
```

```
public static void main(String args[]) {  
    switch (getLivello()) {  
        case ALTO:  
            System.out.println(Livello.ALTO);  
            break;  
        case MEDIO:  
            System.out.println(Livello.MEDIO);  
            break;  
        case BASSO:  
            System.out.println(Livello.BASSO);  
            break;  
    }  
}  
  
public static Livello getLivello() {  
    return Livello.ALTO;  
}
```

8. Se dichiariamo la seguente enumerazione:

```
public enum MyEnum {  
    ENUM1 {  
        public void metodo1() {  
        }  
    },  
    ENUM2 {  
        public void metodo2() {  
        }  
    }  
}
```

il seguente codice potrebbe essere correttamente compilato:

```
MyEnum.ENUM1.metodo1();
```

9. Non è possibile dichiarare enumerazioni con un unico elemento.

10. Si possono innestare enumerazioni in enumerazioni in questo modo:

```
public enum MyEnum {  
    ENUM1 (), ENUM2;  
    public enum MyEnum2 {a,b,c}  
}
```

ed il seguente codice viene compilato senza errori:

```
System.out.println(MyEnum.MyEnum2.a);
```

Soluzioni esercizi modulo 9

Esercizio 9.a) Tipi innestati, Vero o Falso:

- 1.** **Vero.**
- 2.** **Vero.**
- 3.** **Vero.**
- 4.** **Falso**, le classi anonime devono per forza essere istanziate.
- 5.** **Vero**, cfr. paragrafo 9.1.2.
- 6.** **Falso.**
- 7.** **Vero.**
- 8.** **Falso**, una classe anonima non ha nome.
- 9.** **Vero.**
- 10.** **Falso**, una classe anonima non può essere dichiarata astratta.

Esercizio 9.b) Enumerazioni, Vero o Falso:

- 1.** **Vero.**
- 2.** **Vero.**
- 3.** **Falso.**
- 4.** **Falso.**
- 5.** **Vero.**
- 6.** **Falso**, non è possibile utilizzare il costruttore di default se ne viene dichiarato uno esplicitamente.
- 7.** **Vero.**
- 8.** **Vero.**
- 9.** **Vero.**
- 10.** **Vero.**

**Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere cosa sono gli tipi innestati e i vantaggi che comportano (unità 9.1)	<input type="checkbox"/>	
Saper elencare le proprietà fondamentali delle classi innestate (unità 9.1)	<input type="checkbox"/>	
Saper definire ed utilizzare le classi anonime (unità 9.2)	<input type="checkbox"/>	
Comprendere l'utilizzo e l'utilità delle enumerazioni (unità 9.3)	<input type="checkbox"/>	
Comprendere l'utilizzo e l'utilità delle caratteristiche avanzate delle enumerazioni (unità 9.3)	<input type="checkbox"/>	

Note:

Tipi Generici

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Sapere cos'è un tipo generico (unità 10.1).
- ✓ Saper utilizzare i tipi generici (unità 10.1).
- ✓ Comprendere come l'ereditarietà si applica con i tipi generici (unità 10.2).
- ✓ Saper usare le wildcard e i parametri bounded (unità 10.2, 10.3).
- ✓ Saper creare i propri tipi generici (unità 10.3).
- ✓ Saper creare metodi generici (unità 10.3).
- ✓ Comprendere la deduzione automatica del tipo e le conseguenze (unità 10.4).
- ✓ Saper creare parametri covarianti (unità 10.4).
- ✓ Saper gestire i warning del compilatore e le wildcard capture (unità 10.4).

È normale che il codice abbia dei bug. Tuttavia esistono bachi che è possibile risolvere intercettandoli durante la compilazione. Questo si può fare se il linguaggio supporta strumenti come i generici. In particolare i generici oltre a permettere al compilatore di individuare possibili bachi in fase di compilazione, consentono di scrivere codice meno verboso evitando pericolosi cast di tipi, permettendo di creare **algoritmi generici** (ovvero basati sui generici). Si tratta in qualche modo quindi di introdurre un nuovo tipo di programmazione: la **programmazione generica**. Quindi i tipi generici (in inglese generic types) quando furono introdotti nella versione 5 del linguaggio, sconvolsero il modo di scrivere il codice Java. Con i generici la programmazione Java divenne più potente e robusta, ed oggi la libreria standard è basata su tantissime classi, interfacce e metodi generici. L'utilizzo più massiccio dei tipi generici è sicuramente contestuale all'utilizzo delle collection. In questo capitolo quindi introduceremo brevemente qualche collection che ci servirà come esempio, ma le approfondiremo nel modulo dedicato alla libreria `java.util`. L'utilizzo dei generici però, non si limita solo alle collection.

10.1 Generics e tipi parametro

Quando dichiariamo una classe (o un'interfaccia) possiamo “renderla generica” (generic type) aggiungendo alla definizione uno o più tipi parametro (in inglese parameter type). La sintassi fa uso di parentesi angolari che circondano gli identificatori dei tipi:

```
class identificatoreDellaClasse <T1, T2,... Tn> {
    //...
}
```

Per esempio consideriamo la seguente classe, che astrae un contenitore che contiene un oggetto qualsiasi:

```
public class Contenitore {  
    private Object object;  
  
    public void setObject(Object object) {  
        this.object = object;  
    }  
  
    public Object getObject() {  
        return object;  
    }  
}
```

Questa classe sembra molto semplice, ma non è molto facile da gestire. Infatti, una volta recuperato l'oggetto `object` mediante il metodo `getObject()` saremo obbligati a convertirlo per poterlo usare. Questa operazione è però potenzialmente pericolosa, perché potrebbe anche recuperare un tipo diverso da quello che ci aspettiamo. Per esempio, in parti diverse del nostro codice potremmo inserire all'interno di un oggetto di tipo `Contenitore` un oggetto di tipo `Integer`, e poi da qualche altra parte tentare di recuperare una stringa. In casi come questo la Java Virtual Machine lancerà una `ClassCastException`.

In generale per usare questa classe dovremmo scrivere istruzioni come le seguenti:

```
Contenitore contenitore = new Contenitore();  
contenitore.setObject("Stringa");  
// contenitore.setObject(new Integer(1));  
String object = (String)contenitore.getObject();  
System.out.println(object);
```

Si noti che è stato necessario il cast a stringa al momento del recupero dell'oggetto. Se avessimo impostato un oggetto `Integer` nel contenitore (commentando la seconda riga e decommentando la terza riga) avremmo ottenuto una `ClassCastException`.

Per rendere il nostro codice più robusto dobbiamo rendere la classe generica nel seguente modo:

```
public class ContenitoreGenerics<T> {  
    private T object;  
  
    public void setObject(T object) {  
        this.object = object;  
    }  
  
    public T getObject() {  
        return object;  
    }  
}
```

Si noti che ora al posto di `Object` c'è il tipo parametro `T`, che non rappresenta un tipo esistente. Questo verrà sostituito con un tipo reale nel momento in cui verrà istanziato un `ContenitoreGenerics`. Per esempio:

```
ContenitoreGenerics<String> contenitore = new ContenitoreGenerics<String>();  
contenitore.setObject("Stringa");  
String object = contenitore.getObject();  
System.out.println(object);
```

In questo modo è come se avessimo sostituito al parametro `T` la classe `String` in tutta la definizione della classe. Quindi il metodo `setObject()` accetterà solo stringhe (se provassimo a passare altri tipi otterremmo un errore in compilazione) e non c'è bisogno di cast a `String` per recuperare l'oggetto tramite il metodo `getObject()`.

Per convenzione quando si dichiara un tipo parametro si usa un identificatore costituito da una sola lettera maiuscola, che dovrebbe rappresentare l'iniziale di un nome simbolico (nel caso di `T` significa "Type"). In particolare la libreria standard usa spesso: `E` per "Element", `K` per "Key", `N` per "Number", `T` per "Type", `V` per "Value", `S`, `U`, `V` per il secondo, terzo e quarto tipo.

10.1.1 Generics e collection

I Generics offrono la loro più classica utilità quando si usano classi e interfacce del framework `Collection`. Ecco per esempio come è dichiarata l'interfaccia `List`:

```
public interface List<E> extends Collection<E>
```

Come già asserito nel paragrafo precedente, il tipo generico `E` definito per gli oggetti `List`, non è un tipo esistente, ma solo un identificatore generico sostituibile con qualsiasi altro tipo. Per convenzione quando si dichiara un tipo generico si usa un identificatore costituito da una sola lettera maiuscola, che dovrebbe rappresentare l'iniziale di un nome simbolico (`E` dovrebbe significare "Element").

Si noti come l'interfaccia `List` estenda `Collection` dichiarata anch'essa generica.

Siccome `List` è dichiarata generica, quando la si utilizza è possibile parametrizzarla con il tipo che si vuole. Per esempio potremmo istanziare la sua più famosa implementazione: la sottoclassse `ArrayList` (introdotta nella nota conclusiva del paragrafo 7.2.2) con la seguente sintassi:

```
List<Auto> lista = new ArrayList<Auto>();
```

questo significa che potremo aggiungere a questa collezione solo oggetti di tipo `Auto`. Infatti il metodo `add()` che serve per aggiungere elementi alla lista è dichiarato così:

```
public boolean add(E o)
```

In questo modo una volta istanziato l'oggetto `lista` parametrizzandolo con il tipo `Auto`, il tipo parametro `E` diventa di tipo `Auto`, e quindi anche il parametro del metodo `add()` diventa di tipo `Auto`.

Anche se una classe è definita generica, l'utilizzo dei tipi parametri non è obbligatorio. Il compilatore di un IDE però, segnalera un warning in caso di non utilizzo (come vedremo nel paragrafo 10.4.1). Questo per permettere la compatibilità all'indietro con il codice scritto prima dell'avvento di Java 5, dove i generici semplicemente non esistevano.

Supponiamo quindi di aver creato un `ArrayList` in cui vogliamo inserire solo stringhe. Saremo obbligati ad utilizzare un casting quando si estrarranno le stringhe dall'`ArrayList` anche se si conosce a priori il tipo. Per esempio, consideriamo il seguente codice:

```
ArrayList list = getListOfStrings();
int size = list.size();
for (int i = 0; i < size; i++) {
    String stringa = (String)list.get(i);
}
```

Se rimuovessimo il cast, otterremmo un errore in compilazione come il seguente:

```
...
incompatible types
found : java.lang.Object
required: java.lang.String
String stringa = list.get(i);
^
```

Infatti il compilatore non può sapere a priori che tipo di oggetti possono essere stati messi negli elementi dell'`ArrayList`. Per avere codice robusto inoltre, dovremmo comunque garantirci a priori che siano inserite solo stringhe, magari con un controllo come il seguente:

```
if (input instanceof String) {
    list.add(input);
}
```

Come già osservato, senza un controllo come il precedente, potremmo andare incontro ad una delle più insidiose unchecked exception: la `ClassCastException`. Quando usiamo un tipo generico senza specificare i parametri, come abbiamo appena fatto con la classe `ArrayList`, il tipo viene detto raw type. Sarà utile saperlo quando leggeremo i messaggi di warning e di errore del compilatore.

I generici fortunatamente ci permettono di dichiarare una lista specificando che essa accetterà solo stringhe con la seguente sintassi:

```
List<String> strings;
```

Inoltre, bisogna anche assegnare a `strings` un'istanza che accetti lo stesso tipo di elementi (stringhe).

```
List<String> strings = new ArrayList<String>();
```

che da Java 7 in poi possiamo scrivere direttamente così (grazie alla deduzione automatica spiegata nel paragrafo 10.4):

```
List<String> strings = new ArrayList<>();
```

per evitare verbosità superflua.

Le parentesi acute <> sono solitamente chiamate “the diamond”. Anche questa volta diamond dovrebbe essere tradotto come “rombo” e non come “diamante”.

A questo punto abbiamo una lista che accetta solo stringhe e nessun altro tipo di oggetto. Per esempio, il seguente codice:

```
List<String> strings = new ArrayList<>();
strings.add("è possibile aggiungere String");
```

ompilerà tranquillamente, mentre la seguente istruzione:

```
strings.add(new Date());
```

provocherà il seguente output di errore già in compilazione:

```
...
cannot find symbol
symbol : method add(java.util.Date)
location: interface java.util.List<java.lang.String>
strings.add(new Date());
^
```

Quindi i problemi che verranno evidenziati in fase di compilazione ne eviteranno di ben più seri al runtime.

10.1.2 Dietro le quinte

Scendiamo un po' di più nei dettagli. Abbiamo visto come l'interfaccia `List` sia dichiarata nel seguente modo:

```
public interface List<E> extends Collection<E>
```

Abbiamo anche già asserito che è inutile cercare la classe `E`. Si tratta in realtà solo di una nuova terminologia per indicare che `List` è un tipo generico ed `E` rappresenta un parametro. Questo implica che, quando utiliziamo `List`, è possibile parametrizzare `E` con un tipo particolare, come abbiamo fatto nell'esempio.

Anche alcuni metodi hanno cambiato la loro dichiarazione sfruttando parametri. Per esempio, il metodo `add()` è dichiarato nel seguente modo:

```
public boolean add(E o);
```

Possiamo immaginare che nel momento in cui viene istanziato un oggetto e assegnato un reference, il compilatore rimpiazzì tutte le occorrenze di `E` con il tipo specificato tra le parentesi angolari. Infatti, per l'oggetto `strings` dell'ultimo esempio, non viene riconosciuto il metodo `add(E o)` ma `add(String o)`. Ecco spiegato il messaggio di errore.

Questo non avviene per l'intera classe, ma solo per quella particolare istanza trattata. Quindi è possibile creare più liste con differenti parametri. Come nel seguente esempio:

```
List<String> strings = new ArrayList<String>();  
List<Integer> ints = new ArrayList<Integer>();  
List<Date> dates = new ArrayList<Date>();
```

In generale tutte le classi e le interfacce come `Map`, `Iterator`, `Set`, etc. del framework Collections supportano i generici.

10.1.3 Tipi primitivi e autoboxing-autounboxing

I generici non si possono applicare ai tipi di dati primitivi. Quindi il seguente codice:

```
List<int> ints = new ArrayList<int>();
```

restituirà due messaggi di errore del tipo:

```
...  
found : int  
required: reference  
    List<int> ints = new ArrayList<int>();  
    ^  
...
```

Fortunatamente però la seguente sintassi:

```
List<Integer> ints = new ArrayList<Integer>();
```

permetterà tranquillamente di aggiungere interi primitivi. Infatti in Java, grazie alla doppia feature di autoboxing-autounboxing, si possono usare in maniera interscambiabile i tipi primitivi e i relativi "tipi wrapper". Per esempio è possibile scrivere:

```
Integer integer = 1;  
Double d = 3.0D;  
Boolean booleano = true;
```

ma anche:

```
char c = new Character('c');
byte b = new Byte((byte)1);
```

oppure eseguire operazioni:

```
int i = 29;
Short s = new Short ((short)7);
Float f = 74.0F;
double d2 = i*f-7;
```

Approfondiremo il discorso sia sull'autoboxing-autounboxing, sia sulle classi wrapper nel modulo relativo alla libreria `java.lang`.

Per tale ragione è lecito scrivere:

```
ints.add(1);
```

ed anche:

```
int intero = ints.get(0);
```

Si noti che le classi `Byte`, `Short`, `Integer`, `Float`, `Double` e `Long` (ed altre come `BigDecimal`) estendono tutte la classe `Number`.

10.1.4 Interfaccia Iterator

Oltre a `List`, tutte le classi e tutte le interfacce del framework Collections supportano i tipi generici. Più o meno quanto visto per `List` vale per tutte le altre collezioni ed in particolare per `Iterator`. `Iterator` è un'interfaccia molto utilizzata del framework Collections che permette di iterare su una collezione. Solitamente la chiamata al metodo `iterator()` sulla collezione utilizzata, ne restituisce un'implementazione concreta. Astraendo il concetto di iteratore si può definire come un cursore che seleziona in maniera sequenziale gli elementi della collezione. Il metodo `hasNext()` restituisce un booleano, a seconda del fatto che ci siano ancora elementi da processare nella collezione. Il metodo `next()` invece restituisce il prossimo oggetto selezionato dall'iteratore. Si noti che anche `Iterator` è dichiarata generica.

Per esempio, il seguente codice:

```
List<String> strings = new ArrayList<String>();
strings.add("Autoboxing & Auto-Unboxing");
strings.add("Generics");
strings.add("Static imports");
```

```
strings.add("Enhanced for loop");
. . .
Iterator i = strings.iterator();
while (i.hasNext()) {
    String string = i.next();
    System.out.println(string);
}
```

produrrà il seguente output di errore:

```
found : java.lang.Object
required: java.lang.String
    String string = i.next();
1 error
```

Il problema è che bisogna dichiarare anche l'`Iterator` come generico nel seguente modo:

```
Iterator<String> i = strings.iterator();
while (i.hasNext()) {
    String string = i.next();
    System.out.println(string);
}
```

Attenzione a non utilizzare `Iterator` come generico su una `Collection` non generica. Si rischia un'evitabile eccezione al runtime se la `Collection` non è stata riempita come ci si aspetta. D'altronde i generici dovrebbero proprio prevenire questi tipi di problemi per segnalarli in fase di compilazione.

10.1.5 Interfaccia `Map`

Una `Map` è una collezione che associa chiavi ai suoi elementi. Le mappe non possono contenere chiavi duplicate ed ogni chiave può essere associata ad un solo valore. Le mappe sono molto diverse dalle liste. Non sono ordinate come le liste ma consentono un accesso in lettura (grazie alla chiave) più veloce rispetto ad un lista.

L'interfaccia `Map` dichiara due tipi parametro. Segue la sua dichiarazione:

```
public interface Map<K,V>
```

Questa volta i due parametri si chiamano `K` e `V`, rispettivamente iniziali di “Key” (“chiave” in italiano) e “Value” (“valore” in italiano). Infatti, per la mappa possono essere parametrizzati sia le chiavi che i valori. Segue un frammento di codice che dichiara una mappa generica, con parametro chiave di tipo `Integer` e parametro valore di tipo `String`:

```
Map<Integer, String> map = new HashMap<Integer, String>();
```

Grazie all'autoboxing e all'autounboxing sarà possibile utilizzare interi primitivi per valorizzare la chiave. Per esempio, il seguente codice:

```
map.put(0, "generics");
map.put(1, "metadata");
map.put(2, "enums");
map.put(3, "varargs");
for (int i = 0; i < 4; i++) {
    System.out.println(map.get(i));
}
```

inizializza la mappa e ne stampa i valori con un ciclo sulle sue chiavi.

È anche possibile creare “innesti di generici”, per esempio il seguente codice è valido:

```
Map<Integer, ArrayList<String>> map = new HashMap<Integer, ArrayList<String>>();
```

Per ricavare un elemento dell’arraylist innestato è possibile utilizzare il seguente codice:

```
String s = map.get(chiave).get(numero);
```

Dove sia chiave sia numero sono interi. Possiamo notare come non abbiamo utilizzato neanche un cast.

10.2 Ereditarietà dei Generics e Type erasure

Anche i tipi generici formano gerarchie ma tali gerarchie non si basano sui tipi parametri dichiarati. Per esempio il seguente codice è valido:

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
List<Integer> list = arrayList;
```

il seguente codice, invece, non è valido:

```
ArrayList<Number> list = arrayList;
```

Infatti, che Number sia superclasse di Integer non autorizza a considerare il tipo generico ArrayList<Number> superclasse di ArrayList<Integer>. Non è legale neanche la seguente istruzione:

```
ArrayList<Number> list = new ArrayList<Integer>;
```

Il perché sia improponibile che il tipo parametro sia la base di una gerarchia diventa evidente con un semplice esempio. Supponiamo che sia legale il seguente codice:

```
List<Integer> ints = new ArrayList<Integer>();
List<Object> objs = ints;
objs.add("Stringa in un generic di Integer?");
```

Pare ora ovvio che l'ereditarietà non può che essere determinata dal tipo generico. Un'altra ragione per cui l'ereditarietà non si basa sul tipo generico è basata sul concetto di type erasure o semplicemente erasure (in italiano cancellazione del tipo). La gestione dei tipi generici, è gestita solo a livello di compilazione. Ricordiamo che i generici (insieme a tante altre feature come le enumerazioni, le annotazioni, il ciclo for migliorato, i varargs etc.) sono state introdotte nella versione 5 di Java, cercando di impattare il meno possibile su quanto c'era prima. Per rendere possibile l'introduzione dei generici è stato assegnato un ruolo supplementare al compilatore. È il compilatore infatti a trasformare il codice Java in codice pre-Java 5 prima di trasformarlo in bytecode. Quindi, a livello di runtime, le istruzioni:

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
ArrayList<Number> list = arrayList;
```

saranno lette dalla JVM come se fossero stati cancellati (da qui il termine “erasure”) i tipi generici:

```
ArrayList<vector> = new ArrayList();
ArrayList<list> = arrayList;
```

ma a questo punto non si potrebbero più avere informazioni sulla compatibilità degli elementi dei due vettori. Anche per questo il compilatore non permette una ereditarietà basata sui parametri: al runtime i parametri non esistono più.

L'erasure è una delle conseguenze da pagare per la filosofia radicata di Java di essere sempre compatibile con le versioni precedenti. L'obiettivo è permettere agli sviluppatori di poter migrare senza dover buttare tutto il codice scritto in precedenza. Questo non è sempre possibile al 100%. Quindi quando si migra da una versione di Java a quella ...

... successiva, a volte è necessario comunque ritoccare qualche dettaglio del proprio codice. Oracle ha rilasciato questa pagina denominata “Compatibility Guide for JDK 8”:

<http://www.oracle.com/technetwork/java/javase/8-compatibility-guide-2156366.html> dove vengono elencate tutte le incompatibilità tra Java 8 e la versione precedente.

10.2.1 Wildcard

Alla luce di quanto appena visto bisogna fermarsi un attimo per riflettere su quali possono essere le conseguenze dell'erasure. Supponiamo di avere il seguente metodo che fa uso di un raw type:

```
public void print(ArrayList al) {
    Iterator i = al.iterator();
```

```
        while (i.hasNext()) {
            Object o = i.next();
            System.out.println(o);
        }
    }
```

La compilazione di questo codice andrà a buon fine ma verranno segnalati dei warning. Nonostante sia possibile disabilitare i warning con un'opzione di compilazione (`-Xlint:none` cfr. paragrafo 8.5.2), sarebbe sicuramente meglio evitare che ci siano, piuttosto che sopprimerli. La soluzione più semplice è la seguente:

```
public void print(ArrayList<Object> al) {
    Iterator<Object> i = al.iterator();
    while (i.hasNext()) {
        Object o = i.next();
        System.out.println(o);
    }
}
```

Purtroppo però, l'utilizzo del tipo generico, per quanto appena visto sull'ereditarietà nel paragrafo precedente, implicherà semplicemente che questo metodo accetterà in input solo tipi generici con parametro `Object` (e non per esempio `String`). Quello che potrebbe essere un parametro polimorfo, quindi, rischia di essere un clamoroso errore di programmazione.

Ma come risolvere la situazione? A tale scopo esiste una sintassi speciale per i generici che fa uso di wildcard (caratteri jolly) rappresentati da punti interrogativi. In particolare stiamo parlando di unbounded wildcard (in contrasto con le bounded wildcard che vedremo nel paragrafo 10.3.2). Il seguente codice rappresenta l'unica reale soluzione al problema presentato:

```
public void print(ArrayList<?> al) {
    Iterator<?> i = al.iterator();
    while (i.hasNext()) {
        Object o = i.next();
        System.out.println(o);
    }
}
```

Facendo uso di generici, tale metodo non genererà nessun tipo di warning in fase di compilazione ed accetterà qualsiasi tipo di parametro per l'arraylist in input.

Dato che il compilatore non può controllare la correttezza del tipo di parametro quando viene utilizzata una wildcard, rifiuterà di compilare qualsiasi istruzione che tenti di aggiungere o impostare elementi nell'arraylist. In pratica l'utilizzo delle wildcard trasforma i tipi generici “in sola lettura”. Questo significa che all'interno di questo metodo non è possibile modificare l'arraylist.

10.3 Creare propri tipi generici

Come abbiamo già visto è possibile definire i propri tipi generici, come nel seguente esempio:

```

public class OwnGeneric<E> {
    private List<E> list;
    public OwnGeneric () {
        list = new ArrayList<E>();
    }
    public void add(E e) {
        list.add(e);
    }
    public void remove(int i) {
        list.remove(i);
    }
    public E get(int i) {
        return list.get(i);
    }
    public int size() {
        return list.size();
    }
    public boolean isEmpty() {
        return list.size() == 0;
    }
    public String toString() {
        StringBuilder sb = new StringBuilder();
        int size = size();
        for (int i = 0; i < size; i++) {
            sb.append(get(i) + (i != size - 1 ? "-" : ""));
        }
        return sb.toString();
    }
}

```

Probabilmente l'implementazione del metodo `toString()` ha bisogno di qualche osservazione supplementare. Per prima cosa notiamo l'utilizzo della classe `StringBuilder`. Si tratta di una semplice classe particolarmente performante che ha dei metodi per concatenare ed eseguire altre azioni con le stringhe.

Inoltre, l'implementazione del ciclo `for` può risultare criptica a prima vista. In effetti, l'utilizzo dell'operatore ternario non aiuta la leggibilità. Per prima cosa è possibile notare che all'interno del ciclo `for` c'è un'unica espressione. Questa aggiunge una stringa all'oggetto `sb` di tipo `StringBuilder` tramite il metodo `append()`. Il parametro di questo metodo è costituito da quanto restituisce il metodo `get()`, concatenato con il risultato dell'operatore ternario compreso tra parentesi tonde. Tale operatore ritornerà un trattino ("–") per separare i vari elementi estratti dalla collezione, se e solo se il valore di `i` è diverso dalla dimensione della collezione –1. In questo modo l'output risultante avrà una formattazione corretta.

Notiamo come il parametro sia stato definito con una `E` come si usa nella documentazione. Tuttavia, benché sia preferibile utilizzare un'unica lettera maiuscola per definire il parametro, è possibile utilizzare una qualsiasi parola valida per la sintassi Java. Con il seguente codice, invece, andiamo ad utilizzare il nostro tipo generico:

```
OwnGeneric<String> own = new OwnGeneric<String>();
for (int i = 0; i < 10; ++i) {
    own.add(""+ (i));
}
System.out.println(own); //verrà chiamato il metodo toString()
```

L'output sarà:

```
0-1-2-3-4-5-6-7-8-9
```

Per quanto riguarda la classe generica che abbiamo appena creato, non è possibile dichiarare statica la variabile d'istanza `list` nel seguente modo:

```
private static List<E> list = new ArrayList<E>()
```

Infatti questo impedirebbe alle varie istanze della classe di utilizzare parametri differenti (visto che devono essere condivisi).

10.3.1 Parametri e wildcard bounded

È possibile anche creare propri tipi generici con parametri **ristretti** a determinati tipi (bounded parameters). Per esempio se definiamo la classe precedente nel seguente modo:

```
public class OwnGeneric <E extends Number> {
```

potremo utilizzare come parametri solo sottoclassi di `Number` (per esempio `Float` o `Integer`...).

Ora invece supponiamo di voler creare un metodo che prende in input come parametro un tipo generico, il quale a sua volta deve avere un parametro ristretto (bounded) ad un altro tipo. Supponiamo però di non trovarci all'interno di un tipo generico creato da noi, e quindi di non avere un tipo parametro da poter sfruttare. Come possiamo risolvere il nostro problema? La sintassi per implementare tale metodo si può basare su una bounded wildcard:

```
public void print(List <? extends Number> list) {
    for (Iterator<? extends Number> i = list.iterator(); i.hasNext( ); ) {
        System.out.println(i.next());
    }
}
```

L'utilizzo della wildcard è obbligato se non ci troviamo in una classe di tipo generico creata da noi (come la classe `OwnGeneric`) che già dichiara un parametro (nel caso di `OwnGeneric` si chiamava `E`). In questo caso abbiamo usato quella che si dice upper bounded wildcard (ovvero wildcard limitata verso l'alto). Infatti esiste anche la possibilità di usare una wildcard con limite verso il basso (lower bounded wildcard) sfruttando invece della parola chiave `extends` la parola chiave

super. Per esempio il seguente metodo:

```
public static void riempiliLista(List<? super Integer> list) {  
    int size = list.size();  
    for (int i = 1; i <= size; i++) {  
        list.add(i);  
    }  
    System.out.println(list);  
}
```

permette di riempire e stampare una lista riempita di Integer, Number o Object.

Partendo dal fatto che i parametri di un metodo possono essere di tre tipi:

1. parametro che contiene dati da usare (“IN” parameter): esempio del metodo `print()`;
2. parametro che deve essere aggiornato (“OUT” parameter): esempio del metodo `riempiliLista()`;
3. parametro che contiene dati da usare e che deve essere aggiornato.

le linee guida per l’utilizzo delle wildcard sono le seguenti:

- ❑ usare upper bounded wildcard per parametri di tipo 1;
- ❑ usare lower bounded wildcard per parametri di tipo 2;
- ❑ usare unbounded wildcard se la variabile di tipo 1 deve essere usata tramite metodi della classe Object;
- ❑ non usare wildcard per parametri di tipo 3.

In realtà esiste anche una sintassi molto speciale che consente di non utilizzare la wildcard (che abbiamo già avuto modo di notare in un esempio precedente):

```
public <N extends Number> void print(List<N> list) {  
    for (Iterator<N> i = list.iterator(); i.hasNext( ); ) {  
        System.out.println(i.next());  
    }  
}
```

Con l’istruzione:

```
<N extends Number>
```

inserita prima del tipo di ritorno del metodo, stiamo dichiarando localmente un parametro chiamato N che deve avere la caratteristica di estendere Number. Questo parametro sarà utilizzabile all’interno dell’intero metodo. Tale sintassi può essere preferibile quando, per esempio, all’interno del codice del metodo viene spesso utilizzato il parametro <N> (altrimenti siamo obbligati a scrivere più volte “<? extends Number>”). Questo è un esempio di metodo generico.

10.3.2 Metodi generici

I metodi generici non sono altro che metodi che definiscono i propri tipi parametro. La sintassi è la stessa che si usa per i tipi generici, ma la visibilità dei parametri è limitata alla definizione del metodo stesso, sia esso statico, un costruttore o un metodo ordinario. I tipi parametro si devono anteporre al tipo di ritorno del metodo (o al nome del metodo nel caso di metodo costruttore).

Per esempio la seguente classe:

```
public class GenericMethod {
    public static <N extends Number> String getValue(N number) {
        String value = number.toString();
        return value;
    }
    public static void main(String args[]) {
        String value = GenericMethod.getValue(new Integer(25));
        System.out.println(value);
    }
}
```

dichiara il metodo generico `getValue()`, che dichiara un parametro bounded `N` ristretto a `Number` (che quindi potrebbe essere utilizzato con qualsiasi sottoclasse di `Number`). Si noti come anche il parametro `number` in input al metodo è di tipo `N` (che è visibile solo all'interno del metodo `getValue()`). Nel metodo `main()` poi il metodo viene chiamato usando un tipo `Integer` con valore 25 (ma era possibile passargli anche direttamente il valore intero 25).

10.4 Deduzione automatica del tipo

Java 7 ha introdotto un altro piccolo cambiamento di sintassi per la creazione dei tipi generici. La “Type inference for generic instance creation” è una caratteristica che potremmo tradurre come “deduzione automatica del tipo per la creazione di una istanza generica”, ma da ora in poi la chiameremo solamente deduzione automatica (type inference). Dal nome sembrerebbe che si tratti di qualcosa di molto complicato, in realtà è un argomento semplice, a cui tra l'altro, abbiamo già accennato. Come tutti i cambiamenti che hanno influenzato la sintassi di Java 7, anche questo è stato proposto dagli utenti Java e rientrato nel cosiddetto progetto *Project Coin* insieme a tutti gli altri cambiamenti che ci sono stati nella sintassi del linguaggio. Con Project Coin Java si è prefisso di semplificare la vita allo sviluppatore: scrivere meno codice rischiando di sbagliare di meno. Cos'è la deduzione automatica? Facciamo un semplicissimo esempio, consideriamo la seguente dichiarazione:

```
ArrayList<String> arrayList;
```

Per istanziare questo arraylist, prima dell'avvento di Java 7 eravamo costretti a scrivere:

```
ArrayList<String> arrayList = new ArrayList<String>();
```

ora con Dolphin è possibile utilizzare l'operatore “diamond”. In pratica è possibile omettere i parametri per l'oggetto istanziato come nel seguente esempio:

```
ArrayList<String> arrayList = new ArrayList<>();
```

La coppia di parentesi acute vuote viene appunto detta operatore “the diamond”.

Segue un altro esempio più significativo:

```
Map<String, Set<String>> hashMap = new HashMap<String, Set<String>>();
```

può ora essere riscritto come:

```
Map<String, Set<String>> hashMap = new HashMap<>();
```

Insomma perché specificare i tipi parametri nuovamente quando possono essere dedotti dal compilatore automaticamente?

Si noti che possiamo utilizzare l'operatore diamond anche quando si invoca un metodo. Tuttavia questa pratica è sconsigliata, meglio limitarsi ad utilizzarlo nelle dichiarazioni. Per esempio consideriamo il seguente codice:

```
ArrayList<Number> arrayList = new ArrayList<>();
arrayList.add(new Integer(1));
```

compila correttamente . Invece la seguente riga non compila:

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

Infatti, per come è definito il metodo `addAll()` ci si aspetterebbe in input un parametro di tipo `List<E extends Number>`, ed in questo caso il compilatore non può dedurre le intenzioni dello sviluppatore.

Un'altra applicazione avanzata dei tipi generici è quella che permette di utilizzare un parametro per un costruttore, dove questo parametro è diverso da quello che è dichiarato per la classe. Anche in questo caso facciamo un esempio. Consideriamo la seguente classe:

```
public class AdvancedInference<Boolean> {
    public <E> AdvancedInference(E e) {
        System.out.println(e);
    }
}
```

Abbiamo specificato per il costruttore un parametro diverso rispetto a quello definito per la classe (che era di tipo `Boolean`). Nelle versioni precedenti a Java 7 il seguente codice era valido:

```
AdvancedInference<Boolean> test = new AdvancedInference<Boolean>("");
```

Infatti la sintassi `<Boolean>` marcherà il tipo della classe, mentre il valore `String` passato al costruttore andrà ad impostare il parametro `E`, che il costruttore prende in input.

In Java 7 è quindi possibile istanziare questa classe nel seguente modo:

```
AdvancedInference<Boolean> test = new AdvancedInference<>("") ;
```

Infatti l'operatore diamond permetterà la deduzione del parametro della classe (`Boolean`), mentre il valore `String` passato al costruttore andrà ad impostare il parametro `E`, che il costruttore prende in input.

Equivalentemente a questa sintassi, esiste la seguente:

```
AdvancedInference<Boolean> test3 = new <String>AdvancedInference<Boolean>("") ;
```

nella quale si esplicita anche il parametro del tipo del costruttore. Anche se in termini di leggibilità non si tratta di un buon esempio, questa sintassi è valida.

10.4.1 Compilazione

Abbiamo più volte evidenziato che saranno sollevati dei warning se compiliamo file sorgenti che dichiarano collection che potrebbero essere parametrizzate ma non lo sono (raw type). Per esempio, il seguente codice:

```
List strings = new ArrayList();
strings.add("Lambda");
strings.add("Streams API");
strings.add("Date and Time API");
strings.add("JavaFX");
Iterator i = strings.iterator();
while (i.hasNext()) {
    String string = (String)i.next();
    System.out.println(string);
}
```

avrà un esito di compilazione positivo ma genererà il seguente output:

```
Note: Generics1.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Tali warning (o “note”, o “lint” come vengono definiti nelle specifiche) avvertono lo sviluppatore che esistono operazioni non sicure o non controllate, e viene richiesto di ricompilare il file con l'opzione `-Xlint` per avere ulteriori dettagli. Seguendo il suggerimento del compilatore, ricompiliamo il file con l'opzione richiesta. L'output sarà:

```
D:\java8\Codice\modulo_10\esempi\Generics1.java:5: warning: [rawtypes] found
```

```
raw type: List
    List strings = new ArrayList();
    ^
missing type arguments for generic class List<E>
where E is a type-variable:
    E extends Object declared in interface List
D:\java8\Codice\modulo_10\esempi\Generics1.java:5: warning: [rawtypes] found
raw type: ArrayList
    List strings = new ArrayList();
    ^
missing type arguments for generic class ArrayList<E>
where E is a type-variable:
    E extends Object declared in class ArrayList
D:\java8\Codice\modulo_10\esempi\Generics1.java:6: warning: [unchecked] unche
cked call to add(E) as a member of the raw type List
    strings.add("Lambda");
    ^
where E is a type-variable:
    E extends Object declared in interface List
D:\java8\Codice\modulo_10\esempi\Generics1.java:7: warning: [unchecked] unche
cked call to add(E) as a member of the raw type List
    strings.add("Streams API");
    ^
where E is a type-variable:
    E extends Object declared in interface List
D:\java8\Codice\modulo_10\esempi\Generics1.java:8: warning: [unchecked] unche
cked call to add(E) as a member of the raw type List
    strings.add("Date and Time API");
    ^
where E is a type-variable:
    E extends Object declared in interface List
D:\java8\Codice\modulo_10\esempi\Generics1.java:9: warning: [unchecked] unche
cked call to add(E) as a member of the raw type List
    strings.add("JavaFX");
    ^
where E is a type-variable:
    E extends Object declared in interface List
D:\java8\Codice\modulo_10\esempi\Generics1.java:10: warning: [rawtypes] found
raw type: Iterator
    Iterator i = strings.iterator();
    ^
missing type arguments for generic class Iterator<E>
where E is a type-variable:
    E extends Object declared in interface Iterator
D:\java8\Codice\modulo_10\esempi\Generics1.java:19: warning: [cast] redundant
cast to String
    list.add((String)map.get(i));
    ^
8 warnings
```

Questa volta vengono segnalate come warning le righe di codice incriminate con una breve spiegazione nel solito stile Java.

Come è possibile notare questi warning vengono definiti [unchecked]. Esistono vari tipi di warning, di cui gli unchecked sono considerati i più rilevanti. Infatti, compilando con l'opzione -X (fornisce una sinossi delle opzioni non standard che potrebbero cambiare in futuro) scopriremo che riguardo a `Xlint` esistono le seguenti opzioni (usando un compilatore Java 8):

```
-Xlint:{all, auxiliaryclass, cast, classfile, deprecation, dep-ann, divzero, empty, fallthrough, finally, options, overloads, overrides, path, processing, rawtypes, serial, static, try, unchecked, varargs, -auxiliaryclass,-cast,-classfile,-deprecation,-dep-ann, -divzero, -empty, -fallthrough, -finally, -options, -overloads, -overrides, -path, -processing, -rawtypes, -serial, -static, -try, -unchecked, -varargs, none} Enable or disable specific warnings
```

Quindi, se compilando il nostro file con l'opzione `-Xlint` otteniamo troppi warning, possiamo leggere solo la tipologia di warning che ci interessa. Se specifichiamo solo l'opzione `-Xlint` senza specificare sotto-opzioni, di default verrà utilizzata la sotto-opzione `all`.

Per dettagli sulle possibili ulteriori opzioni di `-Xlint` il lettore è invitato a consultare la documentazione dei tool del Java Development Kit (cartella `docs\technotes\tools\index.html` della documentazione ufficiale).

È anche possibile evitare di ricevere warning in compilazione, nel caso si ritenga che sia giusto non utilizzare parametri per i tipi generici. Si potrebbe utilizzare un'annotazione standard chiamata `SuppressWarnings` che dettaglieremo nel modulo 12 relativo proprio alle annotazioni.

La migliore soluzione però è applicare la parametrizzazione ovunque sia possibile.

10.4.2 Parametri covarianti

Quando abbiamo parlato delle regole che governano l'override nel modulo 7 abbiamo accennato ai tipi di ritorno covarianti, ovvero la possibilità di creare override di metodi il cui tipo di restituzione è una sottoclasse del tipo di restituzione del metodo originale. Per esempio, sempre considerando il rapporto di ereditarietà che sussisteva tra le classi `Punto` e `PuntoTridimensionale`, se nella classe `Punto` fosse presente il seguente metodo:

```
public Punto elaboraPunto() {  
    // . . .  
}
```

sarebbe legale implementare il seguente override nella classe `PuntoTridimensionale`:

```
public PuntoTridimensionale elaboraPunto() { // . . .  
}
```

Le regole dell'override però non sono cambiate relativamente ai parametri di input del metodo cui

applicarlo: questi devono coincidere. Nel caso non coincidessero si dovrebbe parlare di overload e non di override (cfr. modulo 7). Non si può parlare quindi dell'esistenza di parametri covarianti in senso stretto. Con l'avvento dei generici però, è possibile implementare un parametro covariante. Stiamo parlando di uno stratagemma (o sarebbe meglio definirlo pattern) non banale. Come al solito, facciamo un esempio. Consideriamo le due seguenti interfacce:

```
interface Cibo {
    String getColore();
}

interface Animale {
    void mangia(Cibo cibo);
}
```

È facile implementare l'interfaccia `Cibo` nella classe `Erba`:

```
public class Erba implements Cibo {
    public String getColore() {
        return "verde";
    }
}
```

come è facile implementare l'interfaccia `Animale` nella classe `Carnivoro`:

```
public class Carnivoro implements Animale {
    public void mangia(Cibo cibo) {
        //un carnivoro potrebbe mangiare erbivori
    }
}
```

e, sia l'interfaccia `Animale` che l'interfaccia `Cibo` nella classe `Erbivoro` (dato che potrebbe essere il cibo di un carnivoro):

```
public class Erbivoro implements Cibo, Animale {
    public void mangia(Cibo cibo) {
        //un erbivoro mangia erba
    }
    public String getColore() {
        ...
    }
}
```

Il problema è che in questo modo sia un carnivoro sia un erbivoro potrebbero mangiare qualsiasi cosa; un carnivoro potrebbe cibarsi d'erba e questo è inverosimile. Potremmo risolvere la situazione sfruttando controlli interni ai metodi che, tramite l'operatore `instanceof` e l'eventuale lancio di un'eccezione personalizzata (`CiboException`) al runtime, impongano i giusti vincoli alle nostre classi. Per esempio:

```

public class Carnivoro implements Animale {
    public void mangia(Cibo cibo) throws CiboException {
        if (!(cibo instanceof Erbivoro)) {
            throw new CiboException("Un carnivoro deve mangiare " +
                "erbivori!");
        }
        . . .
    }
}

public class Erbivoro implements Cibo, Animale {
    public void mangia(Cibo cibo) throws CiboException {
        if (!(cibo instanceof Erba)) {
            throw new CiboException("Un erbivoro deve mangiare + " + "erba!");
        }
        . . .
    }
    public String getColore() {
        . . .
    }
}

public class CiboException extends Exception {
    public CiboException(String msg) { super(msg); }
    . . .
}

```

Inoltre come effetto delle regole dell'override relative alle eccezioni (cfr. modulo 8) per poter compilare correttamente le nostre classi dovremo anche ridefinire l'interfaccia `Animale` in modo tale che il metodo `mangia()` soggetto a override definisca una clausola `throws` per il lancio di una `CiboException`. Segue la ridefinizione dell'interfaccia `Animale`:

```

interface Animale {
    void mangia(Cibo cibo) throws CiboException;
}

```

Le nostre classi sono ora astratte più correttamente ma un eventuale problema verrà rilevato solo durante l'esecuzione dell'applicazione. Per esempio, il seguente codice compilerà correttamente salvo poi sollevare un'eccezione al runtime:

```

public class TestAnimali {
    public static void main(String[] args) {
        try {
            Animale tigre = new Carnivoro();
            Cibo erba = new Erba();
            tigre.mangia(erba);
        } catch (CiboException exc) {

```

```
        exc.printStackTrace();
    }
}
```

Ecco l'output:

```
CiboException: Un carnivoro deve mangiare erbivori!
at Carnivoro.mangia(Carnivoro.java:4)
at TestAnimali.main(TestAnimali.java:6)
```

L'ideale però sarebbe restringere il parametro polimorfo del metodo `mangia()` ad `Erba` per la classe `Erbivoro`, e a `Erbivoro` per la classe `Carnivoro`. In pratica ci piacerebbe utilizzare i parametri covarianti nel metodo `mangia()`. Così infatti, il codice precedente non sarebbe neanche compilabile! E questo sarebbe un vantaggio non da poco. Segue un primo tentativo di soluzione:

```
public class Carnivoro implements Animale {
    public void mangia(Erbivoro erbivoro) {
        ...
    }
}
public class Erbivoro implements Cibo, Animale {
    public void mangia(Erba erba) {
        ...
    }
    public String getColore() {
        ...
    }
}
```

Purtroppo questo codice non compilerà, perché appunto non avremo implementato in nessuna delle due classi il metodo `mangia()` che prende come parametro un oggetto di tipo `Cibo`. Quindi, ereditando metodi astratti in classi non astratte, otterremo errori in compilazione.

Per raggiungere il nostro scopo possiamo però sfruttare appunto i generici. Ridefiniamo l'interfaccia `Animale` parametrizzandola nel seguente modo:

```
interface Animale<C extends Cibo> {
    void mangia(C cibo);
}
```

Ora possiamo ridefinire le classi `Carnivoro` ed `Erbivoro` nel seguente modo:

```
public class Carnivoro implements Animale<Erbivoro> {
    public void mangia(Erbivoro erbivoro) {
        //un carnivoro potrebbe mangiare erbivori
    }
}
```

```

public class Erbivoro<E extends Erba> implements Cibo, Animale<E> {
    public void mangia(E erba) {
        //un erbivoro mangia erba
    }

    public String getColore() {
        ...
    }
}

```

Il codice precedente viene compilato correttamente e impone i giusti vincoli senza problemi per le gerarchie create. Rimangono le evidenti difficoltà di approccio al codice precedente. Segue una classe con metodo `main()` che utilizza correttamente le precedenti:

```

public class TestAnimali {
    public static void main(String[] args) {
        Animale<Erbivoro> tigre = new Carnivoro<Erbivoro>();
        Erbivoro<Erba> erbivoro = new Erbivoro<Erba>();
        tigre.mangia(erbivoro);
    }
}

```

Adesso il metodo `mangia()` dell'interfaccia `Animale` non deve più definire la clausola `throws` per una `CiboException`. Anzi, la classe `CiboException` non è più necessaria.

10.4.3 Cast automatico di reference al loro tipo “intersezione” nelle operazioni condizionali

Una conseguenza del discorso sui generici è il casting automatico dei reference al loro tipo “intersezione”. Stiamo parlando della proprietà che rende compatibili due tipi che hanno una superclasse comune senza obbligare lo sviluppatore ad esplicitare il casting. Per esempio, se consideriamo il seguente codice:

```

ArrayList <Number> list = new ArrayList <Number>();
list.add(1);
list.add(2.0F);
list.add(2.0D);
list.add(2L);
Iterator<Number> i = list.iterator(); while(i.hasNext()) {
    Number n = i.next();
}

```

possiamo notare come non ci sia stato bisogno di esplicitare alcun tipo di casting. Infatti, anche avendo aggiunto a `list` elementi di tipo `Integer`, `Float`, `Double` e `Long` (grazie all'autoboxing), e nonostante il metodo `next()` di `Iterator` restituisca un `Object`, nell'estrazione tramite `Iterator`

parametrizzato con `Number` gli elementi del vettore sono stati automaticamente convertiti a `Number`. Questo perché il compilatore ha svolto il lavoro per noi.

Bisogna quindi osservare un nuovo comportamento dell'operatore ternario (cfr. modulo 4) e più in generale di alcune situazioni condizionali. Precedentemente all'avvento dei generici, infatti, un codice come il seguente provocava un errore in compilazione:

```
import java.util.*;

public class Ternary {
    public static void main(String args[]) {
        String s = "14/04/04";
        Date today = new Date();
        boolean flag = true;
        Object obj = flag ? s : today;
    }
}
```

Segue l'errore evidenziato dal compilatore:

```
incompatible types for ?: neither is a subtype of the other
second operand: java.lang.String
third operand : java.util.Date
    Object obj = flag ? s : today;
    ^
1 error
```

In pratica il compilatore non eseguiva alcun cast esplicito ad `Object`, nonostante sembra scontato che l'espressione non sia ambigua. Il problema si poteva risolvere solo grazie ad un esplicito casting ad `Object` delle due espressioni. Ovvero, bisognava sostituire la precedente espressione riguardante l'operatore ternario con la seguente:

```
Object obj = flag ? (Object)s : (Object)today;
```

Il che sembra più una complicazione che una dimostrazione di robustezza di Java. Nel codice post Java 5 invece, il problema viene implicitamente risolto dal compilatore, senza sforzi del programmatore. Infatti, `s` e `today` vengono automaticamente convertiti ad `Object`.

10.4.4 Wildcard capture e metodi helper

Il seguente codice non compila:

```
public class WildCardTest {
    void test(List<?> list) {
        list.set(0, list.get(0));
    }
}
```

Infatti `list` è parametrizzato con una wildcard ma all'interno del codice del metodo si sta provando a inserire un oggetto, che il compilatore deduce sia un `Object`, in una lista che potrebbe essere parametrizzata al runtime da stringhe come nel seguente esempio:

```
List<String> param = getArrayListCompleto();  
WildCardTest wct = new WildCardTest();  
wct.test(param);
```

Anche se `param` è parametrizzato con stringhe, il compilatore non riesce a dedurre correttamente il tipo parametro, e quindi lo considera `Object`, e un `Object` non può essere messo in una lista che sarà parametrizzata al runtime (nel nostro caso con stringhe). Questo fenomeno di “deduzione limitata” è noto come `wildcard capture`.

Un modo per risolvere la situazione è creare un metodo helper come di seguito:

```
public class WildCardTest {  
    void test(List<?> list) {  
        testHelper(list);  
    }  
    //Metodo helper  
    private <T> void testHelper (List<T> list) {  
        list.set(0, list.get(0));  
    }  
}
```

In questo modo, sfruttando un metodo generico, siamo riusciti a sfruttare il tipo parametro `T` locale, per riuscire nell'assegnazione desiderata.

La tecnica dei metodi helper ha anche una convenzione per il nome del metodo ausiliare, che deve essere del tipo `nomeDelMetodoOriginaleHelper`.

Potremo sfruttare questa tecnica nel momento in cui troveremo in fase di compilazione messaggi di errore che riportano chiaramente errori di “capture”.

Riepilogo

In questo modulo abbiamo trattato uno degli argomenti più importanti di Java: i **tipi generici**. Dopo aver valutato la parametrizzazione di liste, mappe e iteratori con i generici, abbiamo cercato di capire qual è il lavoro del compilatore dietro le quinte. Abbiamo visto come l'ereditarietà non si basi sul tipo parametro e come utilizzare le **wildcard** quando è necessario generalizzare i parametri. Abbiamo anche visto come creare tipi generici personalizzati. Abbiamo anche introdotto la feature nota come **deduzione automatica**. Infine abbiamo valutato l'impatto dei generici sul linguaggio e sul vecchio modo di compilare i nostri file sorgenti oltre ad alcune tecniche per superare problemi tipici della programmazione generica come l'implementazione di **parametri covarianti**, e il superamento dei problemi relativi alle **wildcard capture**.

Esercizi modulo 10

Esercizio 10.a) Generics, Vero o Falso:

1. I tipi generici e i tipi parametro sono la stessa cosa.
2. Un vantaggio principale dei generici è che permettono di evitare bachi come quelli provocati da una `ClassCastException` al runtime, individuandoli in compilazione.
3. Le collection possono essere usate anche senza specificare i tipi parametro. In tal caso si parla di raw type.
4. L'ereditarietà ignora i tipi parametro.
5. Le wildcard si usano quando non abbiamo tipi parametrici a disposizione da usare.
6. Se creiamo un tipo generico con tipo parametro `<E>`, possiamo usare lo stesso parametro anche nei metodi dichiarati nel tipo.
7. Nei metodi generici il tipo parametro viene messo come parametro di input del metodo.
8. Il seguente codice:

```
public class MyGeneric <Pippo extends Number> {  
    private List<Pippo> list;  
    public MyGeneric () {  
        list = new ArrayList<Pippo>();  
    }  
    public void add(Pippo pippo) {  
        list.add(pippo);  
    }  
    public void remove(int i) {  
        list.remove(i);  
    }  
    public Pippo get(int i) {  
        return list.get(i);  
    }  
    public void copy(ArrayList<?> al) {  
        Iterator<?> i = al.iterator();  
        while (i.hasNext()) {  
            Object o = i.next();  
            add(o);  
        }  
    }  
}
```

compila senza errori.

9. Il seguente codice:

```
public <N extends Number> void print(List<N> list) {  
    for (Iterator<N> i = list.iterator();  
         i.hasNext(); ) {
```

```
        System.out.println(i.nextInt());
    }
}
```

compila senza errori.

- 10.** Per risolvere le wildcard capture bisogna usare un metodo helper definito nella libreria standard.

- 11.** Il seguente codice:

```
List<String> strings = new ArrayList<String>();
strings.add(new Character('A'));
```

compila senza errori.

- 12.** Il seguente codice:

```
List<int> ints = new ArrayList<int>();
```

compila senza errori.

- 13.** Il seguente codice:

```
List<int> ints = new ArrayList<Integer>();
```

compila senza errori.

- 14.** Il seguente codice:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
```

compila senza errori.

- 15.** Il seguente codice:

```
List ints = new ArrayList<Integer>();
```

compila senza errori.

Soluzioni esercizi modulo 10

Esercizio 10.a) Generics, Vero o Falso:

- 1. Falso.**, cfr. paragrafo 10.1.
- 2. Vero.**
- 3. Vero.**
- 4. Vero.**
- 5. Vero.**
- 6. Vero.**
- 7. Falso.**
- 8. Falso,** otterremo il seguente errore:

```
(Pippo) in MyGeneric<Pippo> cannot be applied to
(java.lang.Object)
add(o);
^
```

- 9. Vero.**
- 10. Falso,** il metodo helper bisogna crearselo da soli.
- 11. Falso.**
- 12. Falso.**
- 13. Falso.**
- 14. Vero.**
- 15. Vero.**

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Capire cos'è un tipo generico (unità 10.1)	<input type="checkbox"/>	
Saper utilizzare i tipi generici (unità 10.1)	<input type="checkbox"/>	
Comprendere come l'ereditarietà si applica con i tipi generici (unità 10.2)	<input type="checkbox"/>	
Saper usare le wildcard e i parametri bounded (unità 10.2, 10.3)	<input type="checkbox"/>	

Saper creare i propri tipi generici (unità 10.3)	<input type="checkbox"/>
Saper creare metodi generici (unità 10.3)	<input type="checkbox"/>
Comprendere la deduzione automatica del tipo e le conseguenze (unità 10.4)	<input type="checkbox"/>
Saper creare parametri covarianti (unità 10.4)	<input type="checkbox"/>
Saper gestire i warning del compilatore e le wildcard capture (unità 10.4)	<input type="checkbox"/>

Note:

La libreria indispensabile: il package `java.lang`

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Conoscere l'utilizzo e i metodi più importanti della classe `String` (unità 11.1.1).
- ✓ Conoscere l'utilizzo e i metodi più importanti della classe `Object` (unità 11.1.2).
- ✓ Conoscere l'utilizzo e i metodi più importanti della classe `System` (unità 11.1.3).
- ✓ Conoscere l'utilizzo e i metodi più importanti della classe `Runtime` (unità 11.1.4).
- ✓ Conoscere l'utilizzo e i metodi più importanti della classe `Class` e saper sfruttare la tecnica della reflection (unità 11.1.5).
- ✓ Conoscere l'utilizzo delle classi wrapper (unità 11.1.6).
- ✓ Comprendere le semplificazioni che ci offre la (doppia) feature di autoboxing e autounboxing (unità 11.1.7).
- ✓ Conoscere l'utilizzo e i metodi più importanti della classe `Math` (unità 11.1.8).
- ✓ Conoscere le interfacce `Comparable` e `Comparator` per ordinare elementi di array e collezioni (unità 11.1.9).

Sino ad ora abbiamo parlato solo superficialmente delle librerie Java. In realtà sono tantissime, e per coprirle tutte ci vorrebbero decine di libri come questo. Ma non c'è da preoccuparsi, non esiste al mondo alcun programmatore Java che conosca tutte le librerie, è letteralmente impossibile. È invece fondamentale avere sempre a disposizione la documentazione della libreria standard, quando serve qualcosa la si consulta e si trova la soluzione. Esistono però delle classi e delle interfacce che possono essere considerate basilari. Se non si conosce questo minuscolo sottoinsieme della libreria standard non si può dire di avere una cultura Java sufficiente. È arrivato quindi il momento di introdurre il package fondamentale che un programmatore usa praticamente in ogni programma che scrive: il package `java.lang`. Il presente modulo non coprirà tutte le classi presenti in questi package, né lo scopo è quello di sostituire la documentazione ufficiale delle API di Java. Cercheremo piuttosto di capire come utilizzare le classi e le interfacce principali. Inoltre approfondiremo la caratteristica di Java chiamata **autoboxing-autounboxing** che permette di utilizzare i tipi primitivi in luogo di quelli complessi, e introdurremo l'imprescindibile argomento noto come **Reflection**, base di molte tecnologie basate su Java.

11.1 Introduzione al package `java.lang`

Il package `java.lang` oltre a contenere la classe `Thread` e l'interfaccia `Runnable` di cui parleremo ampiamente nel modulo dedicato ai thread, contiene altre decine di classi che già abbiamo

ampiamente utilizzato in questo testo.

La descrizione della documentazione ufficiale descrive così questo package:

Fornisce classi fondamentali per la progettazione del linguaggio Java.

Ricordiamo che stiamo parlando dell'unico package automaticamente importato in ogni programma Java.

In questo modulo introdurremo le classi più utilizzate e famose, con lo scopo di fornire una visione di insieme del package.

11.1.1 La classe `String`

La classe `String` è già stata esaminata più volte in questo testo, in particolare nel modulo 3. Abbiamo già visto come `String` sia l'unica classe che è possibile istanziare come se fosse un tipo di dato primitivo. Inoltre, abbiamo anche visto che tali istanze vengono inserite in un pool di stringhe allo scopo di migliorare le performance, e che tutti gli oggetti sono immutabili.

Osserviamo che questa classe implementa l'interfaccia `CharSequence`. Quando incontreremo metodi che dichiarano come parametri `CharSequence` potremo quindi utilizzare le stringhe grazie al polimorfismo. Inoltre implementa anche l'interfaccia `Comparable`, un'interfaccia (sempre appartenente allo stesso package `java.lang`) che consente un ordinamento di oggetti all'interno di una collection. La classe `String` implementando il metodo ereditato `compareTo()` (spiegato nelle prossime righe) definisce il proprio ordinamento naturale. L'interfaccia `Comparable` sarà approfondita nell'ultimo paragrafo (11.1.9) di questo modulo. Inoltre implementa anche l'interfaccia `Serializable` di cui parleremo nel modulo relativo all'input-output. Di seguito sono elencati i metodi più importanti:

- ❑ `char charAt(int index)` restituisce il carattere all'indice specificato (indice iniziale 0).
- ❑ `String concat(String other)` restituisce una nuova stringa che concatena la vecchia con la nuova (`other`).
- ❑ `int compareTo(String other)` esegue una comparazione lessicale: ritorna un `int < 0` se la stringa corrente è minore della stringa `other`, un intero = 0 se le due stringhe sono identiche e un intero > 0 se la stringa corrente è maggiore di `other`. È l'implementazione del metodo ereditato dall'interfaccia funzionale `Comparable` implementata da `String`.
- ❑ `boolean endsWith(String suffix)` restituisce `true` se e solo se la stringa corrente termina con `suffix`.
- ❑ `boolean equals(Object ob)` cfr. modulo 3.
- ❑ `boolean equalsIgnoreCase(String s)` è un metodo equivalente ad `equals()`, che ignora la differenza tra lettere maiuscole e minuscole.
- ❑ `static String format(String format, Object... args)` permette di formattare una stringa parametrizzandola come è mostrato nel paragrafo 11.2.5. Esiste anche una versione di questo metodo per gestire l'internazionalizzazione (cfr. paragrafo 11.2.2).

- `int indexOf(int ch)` restituisce l'indice del carattere specificato. Si noti che in realtà potremmo passare anche un intero come parametro. Questo perché così sarà possibile anche passare la rappresentazione Unicode del carattere da cercare (per esempio `0XABCD`). Restituisce `-1` nel caso non venga trovato il carattere richiesto. Se esistessero più occorrenze nella stringa del carattere richiesto verrebbe restituito l'indice della prima occorrenza.
- `int indexOf(int ch, int fromIndex)` è equivalente al metodo precedente ma la stringa viene presa in considerazione dall'indice specificato dalla variabile `fromIndex` in poi. Di questi due ultimi metodi esistono versioni con `String` al posto di `int`.
- `static String join(CharSequence delimiter, CharSequence... elements)`: questo metodo statico introdotto in Java 8 ritorna una nuova stringa composta da copie degli elementi del varargs `elements` specificato (ricordiamo che `CharSequence` è implementato da `String`) separati dal delimitatore specificato. Per esempio:

```
String message = String.join("\", \"C:", "Esempi", "Java");
System.out.println(message);
```

stamperà:

```
C:\Esempi\Java
```

Si noti che se un elemento vale `null` allora verrà stampata la stringa `null`.

- `static join(CharSequence delimiter, Iterable<? extends CharSequence> elements)` overload del metodo precedente usa un oggetto che implementa `Iterable` (come un `ArrayList`) di `CharSequence` (per esempio `Stringhe`) in luogo di un varargs come ultimo parametro.
- `int lastIndexOf(int ch)` è come `indexOf()` ma viene restituito l'indice dell'ultima occorrenza trovata del carattere specificato.
- `int length()` restituisce il numero di caratteri di cui è costituita la stringa corrente.
- `String replace(char oldChar, char newChar)` restituisce una nuova stringa dove tutte le occorrenze di `oldChar` sono rimpiazzate con `newChar`.
- `String replace(CharSequence target, CharSequence replacement)` come il metodo precedente ma utilizzando `CharSequence`.
- `String replaceAll(String regex, String replacement)` sostituisce ogni substring di questa stringa che coincide con la regular expression `regex` specificata. Le regular expression (espressioni regolari) saranno spiegate nel modulo relativo alla libreria `java.util`.
- `String replaceFirst(String regex, String replacement)` sostituisce la prima substring di questa stringa che coincide con la regular expression `regex` specificata.
- `String[] split(String regex)` sfruttando la specificata regular expression `regex`, divide la stringa in un array di stringhe.

- ❑ `boolean startsWith(String prefix)` restituisce `true` se e solo se la stringa corrente inizia con la stringa `prefix`.
- ❑ `boolean startsWith(String prefix, int fromIndex)` è equivalente al metodo precedente ma la stringa viene presa in considerazione dall'indice specificato dalla variabile `fromIndex` in poi.
- ❑ `String substring(int startIndex)` restituisce una sottostringa della stringa corrente, composta dai caratteri che partono dall'indice `startIndex` alla fine.
- ❑ `String substring(int startIndex, int endIndex)` restituisce una sottostringa della stringa corrente, composta dai caratteri che partono dall'indice `startIndex` fino all'indice `endIndex - 1`.
- ❑ `String toLowerCase()` restituisce una nuova stringa equivalente a quella corrente ma con tutti i caratteri minuscoli.
- ❑ `String toString()` restituisce la stringa corrente (!), è ereditato dalla classe `Object`.
- ❑ `String toUpperCase()` restituisce una nuova stringa equivalente a quella corrente ma con tutti i caratteri maiuscoli.
- ❑ `String trim()` restituisce una nuova stringa privata dei caratteri spazio, ‘\n’ e ‘\r’, se questi si trovano all'inizio o alla fine della stringa corrente.
- ❑ `static String valueOf(Object obj)` ritorna il valore della chiamata di `toString()` sull'oggetto passato in input oppure `null` nel caso l'oggetto in input sia `null`. Rispetto al chiamare direttamente il metodo `toString()` sull'oggetto, questo metodo ci mette al sicuro da eventuali `NullPointerException`. Di questo metodo viene fatto l'overload per altri tipi di dati primitivi (cfr. documentazione ufficiale).

La classe di utilità `Objects` del package `java.util` fornisce un metodo statico del tutto simile a `valueOf()` di `String`, ma più completo. Infatti prende in input due parametri: il primo è l'oggetto su cui chiamare il metodo `toString()`, il secondo la stringa da ritornare nel caso il primo parametro sia `null`.

Esistono tanti altri metodi ed overload di quelli elencati da consultare direttamente tramite la documentazione ufficiale.

11.1.2 La classe `Object`: metodi `toString()`, `clone()`, `equals()` e `hashCode()`

Abbiamo detto che la classe `Object` è la superclasse di tutte le classi. Ciò significa che, quando codificheremo una classe qualsiasi, eriteremo tutti gli undici metodi di `Object`. Tra questi c'è il metodo `toString()` che avrebbe il compito di restituire una stringa descrittiva dell'oggetto. Nella classe `Object`, che astrae il concetto di oggetto generico, tale metodo non poteva adempiere a questo

scopo, giacché un'istanza della classe `Object` non ha variabili d'istanza che lo caratterizzano. I progettisti di Java decisamente di implementare il metodo `toString()` in modo tale che questo restituisca una stringa contenente informazioni sul reference nel formato: `NomeClasse@IndirizzoInFormatoEsadecimale`, che per la convenzione definita in precedenza potremmo interpretare come **IntervalloDi-Puntamento@IndirizzoInFormatoEsadecimale**. Per esercizio il lettore può provare a stampare un reference qualsiasi con un'istruzione del tipo:

```
System.out.println(nomeDiUnReference);
```

Un altro metodo degno di nota è il metodo `equals()`. Esso è destinato a confrontare due reference (sul primo viene chiamato il metodo e il secondo viene passato come parametro) e restituisce un valore booleano `true` se e solo se i due reference puntano ad uno stesso oggetto (stesso indirizzo di puntamento). Ma questo tipo di confronto, come abbiamo avuto modo di constatare nel modulo 3, è fattibile anche mediante l'operatore `==`. In effetti il metodo `equals()` nella classe `Object` è definito come segue:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

La classe `Object` è troppo generica per poter avere un'implementazione più accurata, per cui in molte sottoclassi di `Object`, come `String`, il metodo `equals()` è stato riscritto in modo tale da restituire `true` anche nel caso di confronto tra due reference che puntano ad oggetti diversi, ma con gli stessi contenuti. Dovrebbe essere questo il vero scopo di un metodo come `equals()`.

In linea teorica la classe `Object` avrebbe dovuto essere astratta e con metodi astratti. Ma dato che ogni classe scritta in Java estende la classe `Object`, questo avrebbe comportato che per ogni nuova classe creata avrebbe dovuto fare override di tutti i suoi metodi astratti... un linguaggio così non avrebbe attirato molti sviluppatori!

Se vogliamo confrontare due oggetti creati da una nostra classe quindi, dovremmo effettuare l'override del metodo `equals()` in modo tale che restituisca `true` se le variabili d'istanza dei due oggetti coincidono. Per esempio, un buon override del metodo `equals()` per la classe `Punto` potrebbe essere la seguente:

```
@Override  
public boolean equals(Object altroOggetto) {  
    if (altroOggetto == null) {  
        return false;  
    }  
    if (altroOggetto instanceof Punto) {  
        Punto altroPunto = (Punto) altroOggetto;  
        return this.x == altroPunto.x && this.y == altroPunto.y;  
    } else {  
        return false;
```

```
}
```

Nella prima riga controlliamo se l'oggetto passato in input è `null`, in tal caso ritorniamo il valore `false`. Dopo viene eseguito un controllo sfruttando l'operatore `instanceof` i cui operandi sono un reference (a sinistra) e una classe (a destra). Nell'esempio quindi entriamo nel blocco di codice dell'`if` se il parametro `obj` è di tipo `Punto`, per poi controllare l'effettiva uguaglianza delle variabili d'istanza. Altrimenti entriamo nella clausola `else` e il metodo `equals()` torna `false`.

Questo codice però è poco elegante ed anche poco chiaro. Esiste una classe di utilità nel package `java.util`, che si chiama `Objects`, che potrebbe aiutarci a semplificare i nostri controlli. Questa mette a disposizione una serie di metodi statici, tra cui per esempio il metodo `isNull()` che restituisce `true` se il parametro di input è `null` o `false` nel caso esso sia diverso da `null`. Il comportamento inverso, è invece definito dal metodo `nonNull()`.

Inoltre è definito anche un metodo `equals()` che prende in input due parametri di tipo `Object` che al suo interno già controlla la nullità degli argomenti nella maniera corretta, così come spiegato per il metodo `isNull()`. Quindi l'esempio precedente potrebbe essere riscritto nel seguente modo:

```
@Override  
public boolean equals(Object altroOggetto) {  
    if (!(altroOggetto instanceof Punto)) {  
        return false;  
    }  
    Punto altroPunto = (Punto) altroOggetto;  
    return Objects.equals(this.x, altroPunto.x) &&  
        Objects.equals(this.y, altroPunto.y);  
}
```

Si noti che nonostante il metodo `equals()` di `Objects` aspetti in input due oggetti di tipo `Object`, è stato possibile passargli direttamente due interi primitivi. Questo grazie all'autoboxing-unboxing (dettagliato più avanti nel paragrafo 11.1.7).

Infine è importante sottolineare che se facciamo override del metodo `equals()` nelle nostre classi, dovremmo anche fare override di un altro metodo: `hashCode()`. Quest'ultimo viene utilizzato silenziosamente da alcune classi (molto importanti come `Hashtable` ed `HashMap`) del framework Collections in maniera complementare al metodo `equals()` per confrontare l'uguaglianza di due oggetti presenti nella stessa collezione. In particolare compilando l'esempio precedente il compilatore con l'opzione `-Xlint` otterremo il seguente warning:

```
D:\java8\Codice\modulo_11\esempi\Punto.java:1: warning: [overrides]  
Class Punto overrides equals, but neither it nor any superclass  
overrides hashCode method  
public class Punto {  
^
```

Un **hash code** non è altro che un `int` che rappresenta l'univocità di un oggetto. Le API della classe `java.lang.Object`, alla descrizione del metodo `hashCode()`, introducono il cosiddetto *contratto* di

`hashCode()`: se due oggetti sono uguali relativamente al loro metodo `equals()` allora devono avere anche lo stesso hash code. Il viceversa non fa parte del contratto: due oggetti con lo stesso hash code non devono essere obbligatoriamente uguali relativamente al loro metodo `equals()`.

Per esempio, un override del metodo `hashCode()` per la classe `Punto` potrebbe essere:

```
public int hashCode() {  
    return x + y;  
}
```

La creazione di un buon metodo `hashCode()` potrebbe anche essere molto più complessa. Per esempio, il seguente metodo è sicuramente più efficiente del precedente per quanto riguarda sia la precisione che le prestazioni:

```
public int hashCode() {  
    return x ^ y;  
}
```

Altri metodi potrebbero risultare più accurati rendendo minore la probabilità di errore (ovvero di collisioni di hash code tra più oggetti), ma bisogna anche tenere conto delle prestazioni. Infatti in alcune situazioni una collezione potrebbe invocare in maniera insistente il metodo `hashCode()`. Un buon algoritmo di hash dovrebbe costituire un compromesso accettabile tra precisione e performance. Dopotutto l'implementazione dipende anche dalle esigenze dell'applicazione. Nonostante la materia degli hash code polinomiali sia vasta e complessa, una convinzione comune (ma non di tutti) è quella che un buon (e semplice) algoritmo di hash coinvolga la moltiplicazione per un numero primo come 31 (ma solitamente insieme a questa moltiplicazione spesso si usano altre operazioni). Per esempio un algoritmo considerato in generale migliore del precedente è il seguente:

```
public int hashCode() {  
    return 31 * (x + y);  
}
```

In generale IDE evoluti come Eclipse e Netbeans consentono la generazione automatica dei metodi `equals()` e `hashCode()`. Eclipse usa proprio una moltiplicazione per 31, mentre Netbeans usa un algoritmo più complesso che coinvolge più di un numero primo.

È importante comunque tenere presente queste osservazioni per non perdere tempo in debug evitabili.

Supponiamo ora che `x` ed `y` siano stati definiti come tipi complessi, per esempio come stringhe. Allora potremmo riscrivere il metodo `hashCode()` nel seguente modo:

```
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result + ((x == null) ? 0 : x.hashCode());
```

```
        result = prime * result + ((y == null) ? 0 : y.hashCode());
    return result;
}
```

Si noti che abbiamo introdotto controlli di nullità mediante l'operatore ternario. Anche in questo caso c'è poca chiarezza e poca eleganza nel nostro codice, ma anche in questo caso ci viene in aiuto la classe `Objects`! Infatti al suo interno troviamo il metodo statico `hashCode()` che prende in input un `Object`, restituendo l'hash code dell'oggetto passato in input oppure 0 se l'oggetto è `null`:

```
public int hashCode() {
    return 31 * (Objects.hashCode(x) + Objects.hashCode(y));
}
```

All'interno di `Objects` troviamo anche il metodo `hash()` che prende in input un varargs di `Object`, e quindi potremmo anche scrivere:

```
public int hashCode() {
    return 31 * Objects.hash(x, y);
}
```

Tornando alla classe `Object`, è definito il metodo `clone()` che restituisce una copia dell'oggetto corrente. La clonazione produce un oggetto della stessa classe dell'originale con le variabili d'istanza contenenti gli stessi valori, e su cui sono permesse modifiche senza che l'oggetto originario venga modificato. A volte clonare un oggetto ci permette di lavorare su una copia senza il rischio di creare qualche problema all'originale. Per garantire che il metodo `clone()` non sia invocato accidentalmente, il metodo è dichiarato `protected`. Essendo dichiarato `protected`, per essere reso disponibile per l'invocazione nelle nostre classi, bisognerà sottoporlo a override cambiandone il modificatore a `public`, come nella seguente classe di esempio:

```
public class CloneableClass implements Cloneable {
    private String campo;
    // Dichiara dei metodi set e get omessi . . .
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Si noti che abbiamo dovuto anche dichiarare la clausola `throws CloneNotSupportedException`, e la nostra classe ha dovuto implementare l'interfaccia `Cloneable`. Infatti, nel caso si chiamasse il metodo `clone()` su una oggetto la cui classe non implementi la suddetta interfaccia, il metodo `clone()` della stessa classe `Object` lancerebbe proprio una `CloneNotSupportedException` producendo il seguente output nel momento in cui viene chiamato il metodo `clone()`:

```
Exception in thread "main" java.lang.CloneNotSupportedException:  
CloneableClass
```

```
at java.lang.Object.clone(Native Method)
at CloneableClass.clone(CloneableClass.java:15)
at TestCloneableClass.main(TestCloneableClass.java:6)
```

Perché tutto questo? Sempre per garantire che il metodo `clone()` non sia invocato accidentalmente. Quindi le seguenti righe:

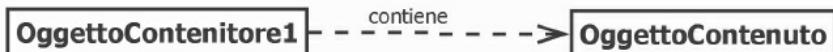
```
CloneableClass cloneableClass = new CloneableClass();
cloneableClass.setCampo("TEST");
System.out.println(cloneableClass);
System.out.println(cloneableClass.clone());
```

Produrranno il seguente output:

```
Oggetto CloneableClass@15db9742 - TEST
Oggetto CloneableClass@6d06d69c - TEST
```

È importante notare che la copia che viene fatta dal metodo `clone()` della classe `Object` si limita a copiare i valori delle variabili d'istanza dell'oggetto (si parla infatti di **shallow copy** ovvero di **copia superficiale**). Quindi se le variabili d'istanza sono reference ad altri oggetti, allora verranno copiati i loro indirizzi, con il risultato che l'oggetto clonato non clonerà anche gli oggetti variabile d'istanza dell'oggetto originale, ma questi saranno in condivisione come mostrato in Figura 11.1.

Prima della clonazione



Dopola clonazione

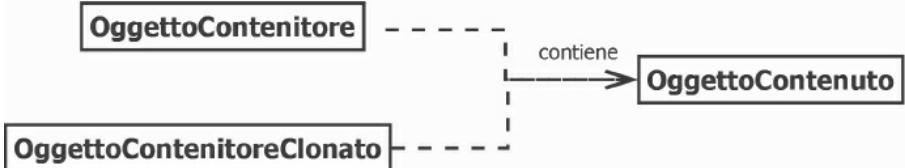


Figura 11.1 - Shallow copy.

Questo tipo di situazione non è desiderabile per una clonazione. Per effettuare una **deep copy** (in italiano **copia profonda**) è necessario riscrivere il metodo `clone()` evitando di chiamare il metodo di `Object` e clonando uno ad uno anche tutte le variabili d'istanza di tipo complesso. Una soluzione percorribile potrebbe anche essere quella di implementare i metodi `clone()` nelle classi delle

variabili d'istanza complesse. Il problema si potrebbe riproporre anche su queste classi, che a loro volta potrebbero dichiarare al proprio interno altri attributi complessi.

Un altro metodo interessante per clonare un oggetto con sicurezza lo vedremo nel capitolo dedicato all'Input-Output.

Nel caso non si invochi più il metodo `clone()` della superclasse `Object` allora non sarà più necessario dichiarare la clausola `throws CloneNotSupportedException`. È comunque buona norma implementare l'interfaccia `Cloneable`.

Esistono altri metodi importanti nella classe `Object` che incontreremo proseguendo nella lettura di questo testo.

11.1.3 La classe `System`

La classe `System` astrae il sistema su cui si esegue il programma Java. Tutto ciò che esiste nella classe `System` è dichiarato statico. Abbiamo più volte utilizzato tale classe quando abbiamo usato l'istruzione per stampare `System.out.println()`. In realtà per stampare abbiamo utilizzato il metodo `println()` della classe `PrintStream` (del package `java.io`). Infatti il metodo è invocato sull'oggetto statico `out` della classe `System` che è di tipo `PrintStream`. L'oggetto `out` rappresenta l'output di default del sistema che dipende dal sistema operativo dove gira l'applicativo.

Esiste anche l'oggetto `err` (`System.err`) che rappresenta l'error output di default del sistema. Anche esso è di tipo `PrintStream` e di solito coincide con l'output di cui sopra. Su un sistema Windows sia l'output di un programma sia gli eventuali errori prodotti dal programma stesso sono visualizzati nella finestra DOS da dove si è eseguito l'applicativo.

Esiste anche un oggetto statico `in` (`System.in`) che astrae il concetto di input di default del sistema. È di tipo `InputStream` (package `java.io` di cui parleremo più avanti in questo testo) e solitamente individua la tastiera del computer.

È possibile modificare il puntamento di queste variabili verso altre fonti di input o di output. Esistono infatti i metodi statici `setOut()`, `setErr()` e `setIn()`. Nel modulo relativo al package `java.io` (input-output) vedremo anche un esempio di utilizzo dell'oggetto `System.in` (leggeremo i caratteri digitati sulla tastiera) e come si può alterare l'indirizzamento di questi oggetti.

In realtà negli esercizi supplementari che trovate on line, un grado di interazione con i nostri programmi è già stato introdotto dai primissimi moduli.

Dopo aver parlato delle variabili membro della classe `System`, diamo uno sguardo ai metodi più interessanti.

Il metodo statico `exit(int code)` consente di bloccare istantaneamente l'esecuzione del programma. Il codice che viene specificato come parametro potrebbe servire al programmatore per capire perché si è interrotto il programma. Un sistema piuttosto rudimentale dopo aver visto la gestione delle eccezioni ed asserzioni. Segue un esempio di utilizzo di questo metodo:

```
if (continua == false) {  
    System.err.println("Si è verificato un problema!");  
    System.exit(0);  
}
```

I metodi `runFinalization()` e `gc()` richiedono alla virtual machine rispettivamente la **finalizzazione** e la **liberazione della memoria occupata** degli oggetti inutilizzati. La finalizzazione consiste nel testare se esistono oggetti non più raggiungibili da qualche reference e quindi non utilizzabili. Questo viene compiuto mediante la chiamata al metodo `finalize()` della classe `Object` (e quindi ereditato in ogni oggetto). Nel caso in cui l'oggetto non sia più reputato utilizzabile dalla Java Virtual Machine viene *segnato* e il successivo passaggio della garbage collection (la chiamata al metodo `gc()`) dovrebbe deallocare la memoria allocata per l'oggetto in questione.

Tuttavia l'utilizzo di questa coppia di metodi è sconsigliato perché la JVM stessa dispone di un meccanismo ottimizzato per gestire il momento giusto in cui chiamare questa coppia di metodi. Per di più, questi due metodi fanno partire dei thread tramite il metodo `start()` che, come vedremo nel modulo relativo ai thread, non garantisce l'esecuzione immediata del metodo stesso.

Altri metodi interessanti sono `setProperty(String key, String value)` e `getProperty(String key)`. Come abbiamo già affermato, la classe `System` astrae il sistema dove viene eseguita l'applicazione. Il sistema possiede determinate proprietà prestabilite. Per esempio:

```
System.out.println(System.getProperty("java.version"));
```

restituirà la versione di Java che si sta utilizzando. È possibile impostare nuove proprietà all'interno della classe `System`, come:

```
System.setProperty("azienda.preferita", "E-Dea S.p.A.");
```

Per poi recuperarle in un secondo momento all'interno di una qualsiasi classe che costituisce il programma. Per un elenco completo di tutte le proprietà disponibili di default all'interno della classe `System` è possibile usare il metodo `getProperties()`. Il seguente codice stampa tutte le proprietà del sistema con i relativi valori:

```
Properties properties = System.getProperties();  
properties.list(System.out);
```

In particolare abbiamo ricavato un oggetto `Properties` mediante il metodo `getProperties()` (che vedremo nel package `java.lang`) e poi abbiamo chiamato su di esso il metodo `list()` che prende in input un `PrintStream` dove andare a stampare. Noi gli abbiamo passato un oggetto `System.out`.

11.1.4 La classe `Runtime`

La classe `Runtime` astrae il concetto di runtime (esecuzione) del programma. Non ha costruttori pubblici ed una sua istanza si ottiene chiamando il metodo factory `getRuntime()`.

Caratteristica interessante di questa classe è permettere di eseguire comandi del sistema operativo

direttamente da Java. Per esempio, il metodo `exec()` (di cui esistono più versioni) utilizzato in questo modo:

```
Runtime r = Runtime.getRuntime();
try {
    r.exec("calc");
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

eseguirà la calcolatrice su un sistema Windows.

Bisogna tener conto che l'utilizzo della classe `Runtime` potrebbe compromettere la portabilità delle nostre applicazioni. Infatti la classe `Runtime` è strettamente dipendente dall'interprete Java e quindi dal sistema operativo. Cercare di eseguire un programma come "calc" va bene su Windows ma non su altri sistemi.

EJE sfrutta proprio la classe `Runtime` per eseguire la compilazione, l'esecuzione, la creazione di javadoc etc., ma il primo prototipo di EJE è stato sviluppato nell'anno 1999. Nella versione 6 di Java è stata aggiunta la possibilità di accedere al compilatore con una nuova libreria, nota come **Java Compiler API**. Sfruttando le classi e le interfacce del package `javax.tools`, con le seguenti righe di codice:

```
JavaCompilerTool compiler = ToolProvider.defaultJavaCompiler();
compiler.run(new FileInputStream("MyClass.java"),null, null);
```

il nostro programma compilerà prontamente la classe `MyClass.java`, stampando eventuali messaggi di errore sulla console.

Esistono tanti altri metodi nella classe `Runtime`. Per esempio il metodo `availableProcessors()` restituisce il numero di processori disponibili. I metodi `freeMemory()`, `maxMemory()` e `totalMemory()` restituiranno informazioni al runtime sullo stato della memoria.

11.1.5 La classe `Class` e Reflection

La classe `Class` astrae il concetto di classe Java. Tra le altre cose questa classe ci permetterà di creare oggetti Java dinamicamente all'interno di programmi Java. Istanziare la classe `Class` si può fare in tre modi diversi:

- utilizzando il metodo statico `forName(String name)` in questo modo:

```
try {
    Class stringa = Class.forName("java.lang.String");
} catch (ClassNotFoundException exc) {
    . . .
}
```

- ottenendola da un oggetto:

```
String a = "MiaStringa";
Class stringa = a.getClass();
```

Si noti che il metodo `getClass()` è un altro metodo definito nella classe `Object`.

- oppure mediante un cosiddetto *class literal*:

```
Class stringa = java.lang.String.class;
```

La caratteristica più interessante di questa classe è la possibilità di utilizzare una tecnica conosciuta come reflection: l'introspezione delle classi. Infatti la classe `Class` mette a disposizione metodi che si chiamano `getConstructor()`, `getMethods()`, `getFields()`, `getSuperClass()` etc., i quali restituiscono oggetti di tipo `Constructor`, `Field`, `Method` e così via. Queste classi astraggono i concetti di costruttore, variabile e metodo e non si trovano all'interno del package `java.lang`, bensì si trovano all'interno del package `java.lang.reflect`. Ognuna di esse definisce metodi per ricavare informazioni specifiche. Per esempio la classe `Field` dichiara il metodo `getModifiers()`.

Inoltre la classe `Class` è generica come vedremo già dal prossimo esempio, che consente di stampare i nomi di tutti i metodi della classe passata da riga di comando (se `TestClassReflection` viene eseguita senza specificare parametri esplora la classe `Object`):

```
import java.lang.reflect.*;

public class TestClassReflection {
    public static void main(String args[]) throws Exception {
        String className = "java.lang.Object";
        if (args.length > 0) {
            className = args[0];
        }
        Class<?> objectClass = Class.forName(className);
        Method[] methods = objectClass.getMethods();
        for (int i = 0; i < methods.length; i++) {
            String name = methods[i].getName();
            Parameter[] classParameters =
                methods[i].getParameters();
            String stringClassParameters = "";
            int length = classParameters.length;
            for (int j = 0; j < length; ++j) {
                stringClassParameters +=
                    classParameters[j].getClass().getName() + " " +
                    classParameters[j].getName();
                // aggiungiamo una virgola per separare i parametri
                // ma solo se non è l'ultimo parametro
                if (j != (length-1)) {
```

```
        stringClassParameters += ", ";
    }
}
String methodReturnType =
methods[i].getReturnType().getName();
String methodString = methodReturnType + " " +
name + "(" + stringClassParameters + ")";
System.out.println(methodString);
}
}
}
```

Nell'esempio è stata utilizzata la classe `Parameter` che è stata introdotta con Java 8, e astrae il concetto di parametro di un metodo. È ora possibile quindi tramite reflection recuperare anche il nome del parametro del metodo. Questa classe dichiara una variabile locale di tipo `String` contenente il nome completo della classe `Object` (in inglese `fully qualified name`) ovvero nome comprensivo di package. Poi però viene controllato se nell'eseguire la classe sia stato specificato un argomento per il comando `java`. Per esempio potremmo eseguire questa classe da riga di comando con:

```
java TestClassReflection java.lang.String
```

in questo modo nella variabile `className` verrà immagazzinato il valore “`java.lang.String`”.

Poi abbiamo istanziato la variabile `objectClass` di tipo `Class<?>`. Abbiamo usato una wildcard perché non sappiamo a priori questa `Class` a che tipo si riferirà (capiremo meglio l'utilizzo del tipo parametro tra qualche riga). Poi in maniera molto intuitiva tramite il metodo `getMethods()` della classe `Class` ci siamo ricavati un array di oggetti `Method`. Con un ciclo su di essi creiamo un algoritmo che stampa le firme dei metodi la cui analisi è lasciata al lettore.

È importante notare che nel bytecode non sono preservati i nomi dei metodi. Se provassimo quindi a eseguire questo programma passandogli una classe compilata da noi, otterremmo al runtime una `NullPointerException` perché i nomi non potranno essere recuperati. Per permettere al compilatore di conservare anche i nomi dei parametri dei metodi, bisogna utilizzare il flag `-parameters` quando si compila la classe da cui leggere i parametri, nel seguente modo:

```
javac -parameters NomeClasse.java
```

Tramite la classe `Class` è anche possibile istanziare oggetti di una certa classe conoscendone solo il nome. Per esempio:

```
try {
    Class miaClasse = Class.forName("MiaClasse");
    Object ref = miaClasse.newInstance();
    . . .
```

```
    } catch (ClassNotFoundException exc) {  
        . . .  
    }
```

Spesso questa tecnica può diventare molto utile, ma attenzione alle stringhe. Il compilatore non ne controllerà la correttezza, quindi un piccolo errore sintattico creerà un baco al runtime.

In realtà la classe `java.lang.Class` è un tipo generico dichiarato nel seguente modo:

```
public final class Class<T> . . .
```

Il parametro `T` rappresenta la classe che verrà usata dall'istanza dell'oggetto `Class` istanziato. Questo ci consentirà di evitare di effettuare cast e sapere a priori cosa contiene un oggetto `Class`. Un semplice esempio di utilizzo di `Class` potrebbe essere il seguente:

```
Class<Punto> puntoClass = Class.forName("Punto");  
Punto punto = puntoClass.newInstance();
```

Si noti che non è stato necessario nessun cast.

La parametrizzazione della classe `Class` permette di scrivere metodi complessi come il seguente:

```
public static <T> T createObjectFromClass(Class<T> type) throws . . . {  
    T object = null;  
    try {  
        object = type.newInstance();  
    } catch (Exception exc) {  
        throw new . . .  
    }  
    return object;  
}
```

Questo metodo (a sua volta parametrizzato per la definizione del parametro `T`) appartenente alla classe `XMVCUtils` del progetto Open Source XMVC (per informazioni <http://sourceforge.net/projects/xmvc>) consente di creare istanze da una classe tramite reflection senza adoperare la tecnica del casting.

Potevamo sostituire a `T` qualsiasi altra lettera (o parola).

Segue un esempio di utilizzo, e come si può osservare è senza cast:

```
MiaClasse miaClasse = XMVCUtils.createObjectFromClass(MiaClasse.class);
```

In questo caso abbiamo parametrizzato anche il metodo `createObjectFromClass()` con il parametro `<T>`.

11.1.6 Le classi wrapper

Sono dette classi wrapper (in italiano *involtuccio*) le classi che fanno da contenitore ad un tipo di dato primitivo, astraendo proprio il concetto di tipo. In Java esistono le classi `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, `Character`, ognuna delle quali può contenere il relativo tipo primitivo. Le classi wrapper sono immutabili come lo è la classe `String`. Questo implica che un oggetto istanziato da una classe wrapper non potrà mai cambiare il suo stato interno tramite un suo metodo, infatti non esistono metodi di tipo "set". Quindi un oggetto `Boolean` istanziato così:

```
Boolean b = new Boolean(true);
```

non conterrà mai il valore `false`.

Queste sono utili ed indispensabili soprattutto nei casi in cui dobbiamo in qualche modo utilizzare un tipo di dato primitivo laddove è richiesto un oggetto. Per esempio, supponiamo di avere il seguente metodo che, tramite un parametro polimorfo di tipo `Object`, definisce l'età di un oggetto di tipo `Persona`:

```
public class Persona {  
    private int age;  
    . . .  
    public void setAge(Object age){  
        // codice che riesce ad impostare l'attributo age  
        // qualunque sia il formato del parametro  
    }  
    . . .
```

L'obiettivo del metodo è definire l'età, che questa sia passata sotto forma di data di nascita o come oggetto `Date`, `Calendar` o `String`. Ora supponiamo che, a seguito di un cambio di requisiti del programma, sia richiesto che questo metodo accetti anche numeri interi rappresentanti il numero di anni piuttosto che la data di nascita. Per come è definito il metodo `setAge()`, supponendo che `piippo` sia un oggetto della classe `Persona`, non è possibile scrivere:

```
piippo.setAge(30);
```

senza ottenere un errore in compilazione. Però è possibile scrivere:

```
Integer anni = new Integer(30);  
piippo.setAge(anni);
```

È necessario poi accomodare il metodo `setAge()` in modo tale che gestisca anche questo nuovo caso, con un codice simile al seguente:

```
public void setAge(Object age){  
    // codice che riesce a impostare la data qualsiasi  
    // sia il formato  
    . . .
```

```
    if (age instanceof Integer) {
        Integer integerAge = (Integer)age;
        this.age = integerAge.intValue();
    }
    . .
}
```

Le classi wrapper `Integer`, `Byte`, `Short`, `Long`, `Character`, `Double` e `Float` (l'unica mancante è la classe `Boolean`) sono tutte sottoclassi di `Number`.

Fondamentali sono i metodi che consentono di fare il parsing da una stringa a un certo tipo di dato. Per esempio `Integer` ha il metodo statico `parseInt()`, che tenta di tradurre in intero una stringa passata in input. Per esempio:

```
int i = Integer.parseInt("1");
```

recupera il valore 1 dalla stringa "1". Siccome abbiamo a che fare con le stringhe bisogna stare attenti ed eventualmente gestire la `NumberFormatException`. Esistono anche i metodi `parseLong()` nella classe `Long`, `parseDouble()` nella classe `Double` e così via. In queste classi esistono ancora tantissimi altri metodi da sfruttare, tra cui il metodo `compareTo()` che abbiamo già trovato nella classe `String`. Infatti le classi wrapper implementano l'interfaccia funzionale e generica `Comparable`, a cui dedicheremo il paragrafo 11.1.9.

Con l'avvento di Java 8 le classi `Integer`, `Byte`, `Short`, `Long`, `Character`, `Double` e `Float` sono state arricchite con i metodi statici `sum()`, `max()` e `min()`, che però vedremo nel modulo dedicato alle collezioni e agli stream. Stesso discorso nella classe `Boolean` alla quale sono stati aggiunti i metodi `logicalAnd()`, `logicalOr()` e `logicalXor()`.

Inoltre è stata anche introdotta la possibilità di usare i tipi senza segno (`unsigned`) mediante il metodo `toUnsignedInt()` e `toUnsignedLong()` per le classi `Byte`, `Short` e `Integer`. Per esempio:

```
byte b = (byte) (128);
System.out.println(Byte.valueOf(b));
System.out.println(Byte.toUnsignedInt(b));
```

stamperà:

```
-128
128
```

Il perché il metodo `valueOf()` stampi -128 è già stato spiegato nel paragrafo 3.2.1, quando abbiamo parlato di tipi primitivi e loro rappresentazione. Invece con il metodo `toUnsignedInt()` stiamo usando l'intervallo degli 8 bit del tipo `byte`, usando solo i numeri positivi (senza segno), quindi l'intervallo di rappresentazione cambia da [-128, 127] a [0, 255].

La classe `Integer` dichiara anche il metodo `toUnsignedString()`, che restituisce sotto forma di stringa il parametro intero in input. È come un metodo `parseInt()` senza segno. Di questo metodo ne esiste anche un'altra versione con un secondo parametro intero, che permette di specificare la base di rappresentazione del numero intero. Per esempio, il seguente frammento di codice in cui sono utilizzate le basi 10 (quella di default), 2, 8 e 16:

```
int i = (int)3000000000L;
System.out.println(Integer.toUnsignedString(i, 10));
System.out.println(Integer.toUnsignedString(i, 2));
System.out.println(Integer.toUnsignedString(i, 8));
System.out.println(Integer.toUnsignedString(i, 16));
```

stampà le rappresentazioni del valore 3000000000 (3 miliardi, che è fuori dall'intervallo di rappresentazione di un intero con segno) in formato decimale, binario, ottale ed esadecimale:

```
3000000000
10110010110100000101111000000000
26264057000
b2d05e00
```

Le classi `Integer` e `Long` inoltre dichiarano metodi per fare operazioni senza segno come `divideUnsigned()`, `remainderUnsigned()` (resto della divisione) e `compareUnsigned()`.

Per fare operazioni senza segno bisogna avere bene in mente come si rappresentano in formato binario i numeri negativi (tecnica del complemento a due). In linguaggi come il C/C++ le operazioni senza segno sono un argomento complesso spesso causa di grossi problemi. Java ha fatto a meno dei tipi senza segno per quasi vent'anni proprio per evitare gli stessi problemi. L'esigenza è dovuta alla difficoltà di usare formati di file e protocolli di rete che richiedono l'aritmetica senza segno, quindi non dovrebbe essere un argomento che useremo spesso in futuro.

11.1.7 Autoboxing-Autounboxing

Un altro tipico esempio in cui è necessario utilizzare le classi wrapper riguarda l'utilizzo delle collezioni. Infatti esse accettano come elementi solo oggetti e non tipi di dati primitivi. Tuttavia grazie all'autoboxing-autounboxing è possibile aggiungere tipi primitivi direttamente alle collezioni. Questa caratteristica permette di fatto di equiparare i tipi primitivi ai relativi tipi wrapper. Quindi sarà possibile tranquillamente aggiungere un tipo primitivo ad una `Collection`, come nel seguente esempio:

```
List list = new ArrayList();
list.add(5.3F);
```

Il compilatore di Java convertirà il codice precedente in modo tale che il valore primitivo 5.3F sia dapprima inscatolato automaticamente (da qui il termine *autoboxing*) nel relativo tipo wrapper (in questo caso un `Float`). Ovvero, è come se il codice dell'esempio precedente fosse trasformato dal compilatore nel seguente codice:

```
ArrayList list = new ArrayList();
list.add(new Float(5.3F));
```

Sarà anche possibile estrarre direttamente il tipo primitivo dalla `Collection` senza alcun passaggio intermedio né cast, grazie all'*autounboxing*.

```
float primitiveFloat = v.get(0);
```

Dal momento che i tipi primitivi ed i relativi tipi wrapper sono di fatto equiparati, sarà possibile eseguire operazioni aritmetiche incrociate come nei seguenti esempi:

```
Integer i = new Integer(22);
int j = i++;
Integer k = (new Integer(10) + j);
int t = k + j + i;
```

Il valore di `t` sarà 77 (tenere presente le regole di priorità degli operatori, cfr. modulo 3).

Il seguente codice:

```
int i = 1;
Integer integer = new Integer(2);
int somma = i + integer;
```

ma anche:

```
Integer somma = i + integer;
```

funziona.

Seguono altri esempi di codice valido:

```
Integer i = 0;
Double d = 2.2;
char c = new Character('c');
```

In definitiva un `int` ed un `Integer` sono equivalenti e così tutti i tipi primitivi con i rispettivi wrapper. Questo favorisce la risoluzione di alcune criticità come l'inserimento di dati primitivi nelle collezioni. Prima di Java 5, per esempio, il codice:

```
ArrayList list = new ArrayList();
list.add(1);
list.add(false);
```

```
list.add('c');
```

avrebbe provocato errori in compilazione. Infatti non era possibile aggiungere un tipo primitivo laddove ci si aspetta un tipo complesso. L'equivalente codice prima di Java 5 doveva essere il seguente:

```
ArrayList list = new ArrayList();
list.add(new Integer(1));
list.add(new Boolean(false));
list.add(new Character('c'));
```

che obbligava a dover recuperare i dati primitivi successivamente, mediante codice di questo tipo:

```
Integer i = (Integer) list.elementAt(0);
Boolean b = (Boolean) list.elementAt(1);
Character c = (Character) list.elementAt(2);
int intero = i.intValue();
boolean booleano = b.booleanValue();
character c = c.charValue();
```

Ma perché fare tanta fatica?

Ora è possibile scrivere direttamente:

```
ArrayList list = new ArrayList();
list.add(1);
list.add(false);
list.add('c');
int intero = (Integer) list.elementAt(0);
boolean booleano = (Boolean) list.elementAt(1);
character c = (Character) list.elementAt(2);
```

e questo non può che semplificare la vita del programmatore.

Il termine “boxing” equivale al termine italiano “inscatolare”. L’inscatolamento dei tipi primitivi nei relativi tipi wrapper è automatico (ci pensa il compilatore). Ecco perché si parla di autoboxing. L’autounboxing invece è il processo inverso. Si tratta quindi di una doppia funzionalità. Il compilatore non fa altro che inscatolare i tipi primitivi nei relativi tipi wrapper, o estrarre tipi primitivi dai tipi wrapper, quando ce n’è bisogno. L’autoboxing e l’autounboxing sono comodità forniteci dal compilatore che svolge del lavoro per noi.

Le specifiche di Java asseriscono che alcuni tipi di dati primitivi sono sempre inscatolati nelle stesse istanze wrapper immutabili. Tali istanze sono poi poste in una speciale cache dalla Virtual Machine e riusate, perché ritenute di frequente utilizzo. Tutto questo col fine di migliorare le performance. Godono di questa caratteristica:

1. tutti i tipi `byte`;
2. i tipi `short` e `int` con valori compresi nell'ordine dei `byte` (da -128 a 127);
3. i tipi `char` con range compreso da \u0000 a \u007F (cioè da 0 a 127);
4. i tipi `boolean`.

Vedremo presto che, benché questa osservazione sembri fine a se stessa, ha un'importante conseguenza di cui i nostri programmi devono tener conto.

La seguente istruzione è illegale:

```
Double d = 2;
```

Infatti il valore 2 è di tipo `int`. L'autoboxing richiede che ci sia una coincidenza perfetta tra il tipo primitivo e il relativo tipo wrapper. Non importa che un `int` possa essere contenuto in un `double`. Il problema è facilmente risolvibile con il seguente cast (cfr. modulo 3):

```
Double d = 2D;
```

11.1.7.1 Assegnazione di un valore `null` al tipo wrapper

Non sempre l'autoboxing-autounboxing è così semplice. È sempre possibile assegnare il valore `null` ad un oggetto della classe wrapper. Ma bisogna tener presente che `null` non è un valore valido per un tipo di dato primitivo. Quindi il seguente codice:

```
Boolean b = null;  
boolean bb = b;
```

compila tranquillamente, ma lancia una `NullPointerException` al runtime. Questo è forse il principale problema che si può incontrare utilizzando l'autoboxing e l'autounboxing.

11.1.7.2 Costrutti del linguaggio ed operatori relazionali

Tutti i costrutti del linguaggio (`if`, `for`, `while`, `do`, `switch` e l'operatore ternario) e tutti gli operatori di confronto (`<,>`, `==`, `!=`, etc.) si basano sui tipi booleani. Con Java potremo sostituire un tipo `boolean` con il rispettivo tipo wrapper `Boolean` senza problemi, anche in tutti i costrutti e operatori. Il costrutto `switch` (cfr. paragrafi 4.4.4 e 9.3.6) inoltre, si basa su una variabile di test che può essere di tipo enumerazione, `String`, `byte`, `short`, `int` o `char`. Alla luce di quanto appena visto quindi, tale variabile potrà essere anche di tipo `Byte`, `Short`, `Integer` o `Character`.

L'operatore `==` è utilizzabile per confrontare sia tipi di dati primitivi, sia reference. In ogni caso va a confrontare i valori delle variabili coinvolte che, nel caso di tipi reference, sono indirizzi in

memoria (cfr. modulo 4). Questo implica che due oggetti confrontati con l'operatore == saranno uguali se e solo se risiedono allo stesso indirizzo, ovvero se sono lo stesso oggetto. L'introduzione della doppia funzionalità di autoboxing e autounboxing ci obbliga ad alcune riflessioni. Visto che i tipi primitivi e i tipi wrapper sono equivalenti, che risultati darà l'operatore ==? Il risultato è quello che ci si aspetta sempre, le regole non sono cambiate. Ma c'è un'eccezione. Consideriamo il seguente codice:

```
public class Comparison {  
    public static void main(String args[]) {  
        Integer a = 1000;  
        Integer b = 1000;  
        System.out.println(a==b);  
        Integer c = 100;  
        Integer d = 100;  
        System.out.println(c==d);  
        int e = 1000;  
        Integer f = 1000;  
        System.out.println(e==f);  
        int g = 100;  
        Integer h = 100;  
        System.out.println(g==h);  
    }  
}
```

L'output del precedente programma sarà:

```
false  
true  
true  
true
```

Ovvero, tranne che nella comparazione tra `c` e `d`, tutto funziona in maniera normale. Infatti `c` e `d` sono due `Integer` con valori compresi nel range del tipo `byte` (-128 a 127) e quindi, come abbiamo asserito nel paragrafo precedente, vengono trattati in maniera speciale dalla JVM. Questi due `Integer` vengono inscatolati nelle stesse istanze wrapper immutabili, per essere memorizzate e riusate. Quindi i due `Integer` risultano condividere lo stesso indirizzo perché in realtà sono lo stesso oggetto.

L'ultima considerazione è da considerarsi indubbiamente un problema, ma se scriviamo codice che utilizza il metodo `equals()` con i tipi complessi e l'operatore == con i tipi primitivi questo problema non si presenterà mai. In fondo, questa dovrebbe essere la regola da seguire sempre.

11.1.7.3 Overload

L'overload (cfr. modulo 7) è una caratteristica di Java molto semplice e potente. Grazie ad essa è possibile far coesistere in una stessa classe più metodi con lo stesso nome ma con differente lista di

parametri. Prima dell'avvento dell'autoboxing era molto semplice capire la chiamata ad un metodo sottoposto a overload. La lista dei parametri non generava nessun dubbio. Ma consideriamo il seguente codice:

```
public void metodo(Integer i) { . . . }
public void metodo(float f) { . . . }
```

Quale metodo verrà chiamato dalla seguente istruzione?

```
metodo(123);
```

La risposta è semplice: lo stesso che veniva chiamato con le versioni precedenti (ovvero `metodo(float f)`). In questo modo è stata evitata una scelta che avrebbe avuto conseguenze inaccettabili per gli sviluppatori, ovvero sarebbe stata negata la compatibilità con il codice preesistente.

11.1.8 La classe `Math`

Questa classe appartiene al package `java.lang` (da non confondere con il package `java.math`). Ha la caratteristica di contenere solo membri statici: due costanti (**pi greco** e **e** base dei logaritmi naturali) ed oltre trenta metodi. I metodi rappresentano:

- ❑ le funzioni matematiche valore assoluto (`abs()`), tangente (`tan()`), logaritmo (`log()`), potenza (`pow()`), massimo (`max()`), minimo (`min()`), seno (`sin()`), coseno (`cos()`), esponenziale (`exp()`) e radice quadrata (`sqrt()`);
- ❑ arrotondamenti per eccesso (`ceil()`), per difetto (`floor()`) e classico (`round()`);
- ❑ generazione di numeri casuali (`random()`);

Questa classe non ha un costruttore pubblico ma privato, quindi non si può istanziare. Non è nemmeno possibile estenderla, per due ragioni:

1. ha un costruttore privato.
2. È dichiarata `final`.

Un esempio di utilizzo della classe `Math` è il seguente:

```
System.out.println(Math.floor(5.6));
```

In questo caso verrà stampato il valore `5.0`. Infatti la “funzione” `floor()` restituisce un numero `double` che arrotonda per difetto il valore del parametro in input sino al più vicino numero intero.

Con Java 8 sono stati introdotti nuovi metodi del tipo **exact**. Questi metodi lanciano un’eccezione nel caso vengano sfornati i limiti numerici del tipo (overflow). Questo significa che quando per esempio `100*1000000000` in Java restituisce il valore errato `1215752192` (cfr. modulo 3), il metodo `Math.multiplyExact(100, 1000000000)` lancerà un’eccezione:

```
Exception in thread "main" java.lang.ArithmeticException: integer overflow
at java.lang.Math.multiplyExact(Math.java:867)
at TestMathExact.main(TestMathExact.java:4)
```

Esistono i metodi `addExact()`, `subtractExact()`, `negateExact()`, `multiplyExact()`, `incrementExact()`, `decrementExact()`.

11.1.9 Ordinamento: le interfacce Comparable e Comparator

L'interfaccia funzionale `Comparable` definisce un unico metodo astratto: `compareTo()`. È implementata da centinaia di classi delle librerie standard. Abbiamo già incontrato infatti il metodo `compareTo()` nel paragrafo 11.1.1 quando abbiamo esplorato i metodi della classe `String`, e nel paragrafo 11.1.6 quando abbiamo parlato di classi wrapper. Infatti sia `String` che le classi wrapper implementano l'interfaccia `Comparable`. Essa è anche generica. Ecco la sua definizione epurata dai commenti:

```
package java.lang;
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Per esempio la classe `String` è stata definita nel seguente modo:

```
public final class String implements Comparable<String> // . . .
```

Il fatto di aver utilizzato nella definizione della classe `String` come tipo parametralo per `Comparable` il tipo `String`, implica che il metodo `compareTo()` accetterà solo stringhe. Con la stessa logica sono state definite tutte le classi che implementano l'interfaccia `Comparable` (comprese le classi wrapper), per garantire che l'ordinamento sarà fatto solo tra oggetti istanziati dalla stessa classe.

Quindi il metodo `compareTo()` viene chiamato su un oggetto e gli viene passato un oggetto della stessa classe, così il confronto all'interno del metodo viene fatto tra l'oggetto corrente e l'oggetto passato in input. Il contratto del metodo `compareTo()`, dice che il metodo deve ritornare il valore `-1` se l'oggetto corrente è minore dell'oggetto in input. Deve ritornare `0` se i due oggetti sono identici. Infine deve tornare il valore `+1` nel caso l'oggetto corrente sia maggiore dell'oggetto in input. Quindi il seguente frammento di codice:

```
Integer int1 = 4;
Integer int2 = 5;
Integer int3 = 5;
System.out.println(int1.compareTo(int2));
System.out.println(int2.compareTo(int1));
System.out.println(int2.compareTo(int3));
```

produrrà il seguente output:

```
-1  
1  
0
```

Ma come e quando si usa l'ordinamento? In generale quando una classe implementa Comparable significa che è ordinabile all'interno di una collezione o un array. Per esempio se vogliamo ordinare un array di stringhe, avremo a disposizione il metodo `sort()` della classe di utilità `Arrays` (che si trova nel package `java.util`) :

```
String [] nomi = {"Marcello", "Serena", "Antonio"};  
System.out.println("Array non ordinato:");  
stampaArray(nomi);  
Arrays.sort(nomi);  
System.out.println("\nArray ordinato:");  
stampaArray(nomi);
```

dove il metodo `stampaArray()` stampa con un ciclo `foreach` gli elementi dell'array. Il codice precedente produrrà il seguente output:

```
Array non ordinato:  
Marcello  
Serena  
Antonio  
  
Array ordinato:  
Antonio  
Marcello  
Serena
```

Esiste anche la classe `java.util.Collections` che rappresenta la classe di utilità per le collezioni, come `Arrays` lo è per gli array. Queste classi contengono anche tanti altri metodi di utilità per lavorare su collezioni ed array. Torneremo sull'argomento nel Modulo 16.

Esiste anche un'altra interfaccia molto simile a Comparable, si chiama Comparator e non appartiene al package `java.lang` bensì al package `java.util`. Tuttavia ci sembra opportuno completare il discorso sull'ordinamento in questo paragrafo. L'interfaccia Comparator è anch'essa funzionale e generica. Segue la sua dichiarazione semplificata:

```
package java.util;  
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    ...
```

La differenza tra il metodo `compare()` di Comparator e il metodo `compareTo()` di Comparable visto prima, è che `compare()` viene chiamato su un'istanza di Comparator e prende in input le due istanze della classe (che deve corrispondere al tipo generico) che bisogna comparare. Per esempio potremmo creare una classe che implementa Comparator che ordina le stringhe dopo averle

```
import java.util.Comparator;

public class StringComparator implements Comparator<String> {
    public int compare (String s1, String s2) {
        StringBuilder sb1 = new StringBuilder(s1);
        StringBuilder sb2 = new StringBuilder(s2);
        s1 = sb1.reverse().toString();
        s2 = sb2.reverse().toString();
        return s1.compareTo(s2);
    }
}
```

Si noti che abbiamo usato il metodo `reverse()` per capovolgere le stringhe dopo averle messe all'interno di istanze di `StringBuilder`, una classe d'utilità di `java.lang`, che consente alcune operazioni sulle stringhe.

A questo punto consideriamo per esempio il seguente frammento di codice:

```
List<String> nomiList = new ArrayList<String>(Arrays.asList(nomi));
System.out.println("List non ordinata:");
stampaList(nomiList);
Collections.sort(nomiList, new StringComparator());
System.out.println("List ordinata a stringhe al contrario:");
stampaList(nomiList);
```

dove abbiamo usato il metodo generico `asList()` di `Arrays` per riempire la lista con gli elementi dell'array del precedente frammento di codice. Inoltre abbiamo usato un overload del metodo `sort()` che prende in input come parametro un oggetto `Comparator`, permettendoci di ordinare secondo il nuovo ordine stabilito dalla classe `StringComparator` da noi creata. L'output del precedente frammento di codice sarà:

```
List non ordinata:
Antonio
Marcello
Serena
List ordinata a stringhe al contrario:
Serena
Antonio
Marcello
```

Riepilogo

In questo modulo abbiamo introdotto essenzialmente le classi principali del package `java.lang`, con qualche digressione su classi di altri package, come `Objects`, `Arrays`, `Collections` e `Comparator`, del package `java.util`.

Abbiamo prima ripreso la classe `String` mostrando una panoramica dei metodi più importanti. Poi siamo passati a guardare i metodi principali della classe `Object`, ed in particolare abbiamo affrontato anche l'argomento **clonazione**, oltre ai metodi `toString()`, `equals()` e `hashCode()`. La classe `Object` è particolarmente importante per il fatto che è la superclasse di tutte le classi, e quindi capita spesso di fare override dei suoi metodi. Le classi **wrapper** invece rappresentano un importantissimo strumento quando si utilizzano le collezioni e non solo. Con l'**autoboxing** però è ora possibile utilizzare tipi wrapper e tipi primitivi quasi allo stesso modo. Nonostante questa funzionalità sia particolarmente semplice da utilizzare, nasconde comunque alcune insidie come è stato mostrato.

La **Reflection** rappresenta uno strumento essenziale per applicazioni come EJE e tante altre. La possibilità di creare oggetti a partire da stringhe (nome della classe) è particolarmente importante, e su di essa si basano molte tecnologie e framework moderni. Inoltre sono state introdotte le classi `Math`, `System` e `Runtime`, tutti argomenti molto semplici che impareremo ad usare molto presto. Infine abbiamo anche affrontato il discorso dell'**ordinamento**, che può essere realizzato mediante l'implementazione dell'interfaccia funzionale `Comparable` (da implementare direttamente nella classe degli oggetti da ordinare) o alternativamente dall'implementazione dell'interfaccia funzionale `java.util.Comparator` (che può essere implementata in una classe qualsiasi).

Esercizi modulo 11

Esercizio 11.a) Autoboxing, autounboxing e `java.lang`. Vero o Falso:

- Il seguente codice compila senza errori:

```
char c = new String("Uappi");
```

- Il seguente codice compila senza errori:

```
int c = new Integer(1) + 1 + new Character('a');
```

- Le regole dell'overload non cambiano con l'introduzione dell'autoboxing e dell'autounboxing.

- Le istanze della classe `Integer` sono immutabili, e quindi non è possibile mutare il loro stato interno una volta istanziate.

- La classe `Runtime` dipende strettamente dal sistema operativo su cui esegue.

- La classe `Class` permette di leggere i membri di una classe (ma anche le superclassi ed altre informazioni) partendo semplicemente dal nome della classe grazie al metodo `forName()`.

- Mediante la classe `Class` è possibile istanziare oggetti di una certa classe conoscendone solo il nome.

- Dalla versione 1.4 di Java è possibile sommare un tipo primitivo e un oggetto della relativa classe wrapper, come nel seguente esempio:

```
Integer a = new Integer(30);
int b = 1;
```

```
int c = a+b;
```

9. La clonazione di oggetti richiede obbligatoriamente una chiamata al metodo `clone()` di `Object`.
10. La classe `Math` non si può istanziare perché dichiarata `abstract`.

Soluzioni esercizi modulo 11

Esercizio 11.a) Autoboxing, autounboxing e `java.lang`, Vero o Falso:

1. **Falso.**
2. **Vero.**
3. **Vero.**
4. **Vero.**
5. **Vero.**
6. **Vero.**
7. **Vero.**
8. **Falso**, dalla versione 1.5.
9. **Falso.**
10. **Falso**, non si può istanziare perché ha un costruttore privato ed è dichiarata `final` per non poter essere estesa.

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Conoscere l'utilizzo e i metodi più importanti della classe <code>String</code> (unità 11.1.1)	<input type="checkbox"/>	
Conoscere l'utilizzo e i metodi più importanti della classe <code>Object</code> (unità 11.1.2)	<input type="checkbox"/>	
Conoscere l'utilizzo e i metodi più importanti della classe <code>System</code> (unità 11.1.3)	<input type="checkbox"/>	
Conoscere l'utilizzo e i metodi più importanti della classe <code>Runtime</code> (unità 11.1.4)	<input type="checkbox"/>	
Conoscere l'utilizzo e i metodi più importanti della classe <code>Class</code> e saper sfruttare la tecnica della reflection (unità 11.1.5)	<input type="checkbox"/>	
Conoscere l'utilizzo delle classi wrapper (unità 11.1.6)	<input type="checkbox"/>	
Comprendere le semplificazioni che offre la (doppia) funzionalità di autoboxing e autounboxing (unità 11.1.7)	<input type="checkbox"/>	

Conoscere l'utilizzo e i metodi più importanti della classe <code>Math</code> (unità 11.1.8)	<input type="checkbox"/>
Conoscere le interfacce <code>Comparable</code> e <code>Comparator</code> per ordinare elementi di array e collezioni (unità 11.1.9)	<input type="checkbox"/>

Note:

Tipi annotazioni

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Comprendere cosa sono i metadati e la loro relatività (unità 12.1, 12.2).
- ✓ Comprendere l'utilità delle annotazioni (unità 12.1, 12.2, 12.3).
- ✓ Saper definire nuove annotazioni (unità 12.1, 12.2).
- ✓ Saper annotare elementi Java ed altre annotazioni (unità 12.2, 12.3).
- ✓ Saper utilizzare le annotazioni definite dalla libreria: le annotazioni standard e le meta-annotazioni (unità 12.3, 12.4).
- ✓ Comprendere e saper utilizzare le novità di Java 8 per le annotazioni come le repeating annotation, le type annotation e nuove annotazioni standard come `FunctionalInterface` (unità 12.2, 12.3)

Questo modulo spiega un argomento sicuramente non semplice. La difficoltà di formulare correttamente le definizioni essenziali, la complessità di una sintassi non standard, il livello elevato di astrazione dal linguaggio, l'uso di una nuova parola chiave speciale, l'introduzione di un nuovo tipo di paradigma di programmazione e la relativa applicabilità delle annotazioni, richiederanno al lettore un livello di concentrazione elevato. In compenso capiremo perché le annotazioni rappresentano potenzialmente una delle caratteristiche più importanti di Java. In questo modulo non parleremo di programmazione standard fatta da `if`, `for` e paradigni object oriented, ma di meta-programmazione. In ogni caso, riteniamo utile avvertire il lettore meno esperto che non sarà facile apprezzare alcuni aspetti di questo argomento. È improbabile per esempio che si definiscano da subito le proprie annotazioni e i propri processori di annotazioni. Ma esistono alcune parti senz'altro più abbordabili (vedi annotazioni standard) che potrebbero risultare utili da subito. In effetti molti programmatore Java reputano quest'argomento semplice e secondario. Si limitano a saper utilizzare le classiche annotazioni definite nei framework web ed enterprise come Enterprise Java Beans, Hibernate, Struts etc. In fondo il programmatore le utilizza e qualche tool le processa, niente di più semplice! In realtà però, le annotazioni sono molto più di questo. La prima parte di questo modulo sarà dedicata alla definizione delle annotazioni. È abbastanza complessa, ma importante. Infatti impareremo a creare le nostre annotazioni e ne studieremo la sintassi. La seconda parte del modulo è dedicata allo studio delle annotazioni standard, che potranno risultare utili da subito anche al programmatore meno esperto. Ne approfitteremo per analizzare anche aspetti più complessi che potrebbero far annoiare il neofita, ma illuminare lo sviluppatore esperto.

12.1 Definizione di annotazione (metadato)

Con il termine **metadati**, solitamente si intendono le **informazioni sulle informazioni**. La frase precedente non è poi così chiara, ma dovremmo filosofeggiare un po' per poter dare una definizione corretta. Quindi cercheremo di capire il concetto con esempi prima di entrare nel cuore dell'argomento. Nel Modulo 2 abbiamo definito il concetto di classe come "*un insieme di oggetti che condividono le stesse caratteristiche e le stesse funzionalità*". Ma proviamo a guardare l'argomento da un'altra angolazione. Proviamo a pensare cos'è una classe per il compilatore o la virtual machine. Passiamo subito a fare un esempio. Se volessimo spiegare a qualcuno in lingua italiana cos'è una persona, cosa ovviamente molto complessa, utilizzeremmo una serie di frasi del tipo "ha un nome, un cognome, un'età, etc.". In pratica definiremmo il concetto di persona tramite l'elenco delle sue caratteristiche e delle sue funzionalità. Tutto questo va bene, ma è comprensibile solo se si hanno già in mente le definizioni di nome, cognome, età e così via. Ovvero, se il nostro interlocutore non sa cosa è un nome, potrebbe pensare che un nome valido sia "831". Ugualmente, se non ha mai sentito parlare di età, potrebbe pensare che un'età possa avere questa valore: "- 99". Ecco allora il bisogno dei metadati: le informazioni sulle informazioni. Se definiamo un'età come un numero intero di anni non minore di zero, diventa tutto più chiaro.

Per qualcuno l'esempio precedente potrebbe essere contestabile. Nel mondo reale non bisogna (fortunatamente) definire per forza tutti i metadati per farsi capire, almeno . . .

. . . in una situazione così banale. Ma se volessimo definire cosa è per la geometria differenziale la "superficie romana di Steiner" (uno degli argomenti della mia tesi di laurea in matematica), non credo che senza una quantità notevole di metadati potremmo capirci.

Molto spesso i metadati non sono altro che **vincoli** (in inglese *constraints*). Per esempio, nella definizione di età abbiamo specificato come vincolo che il numero degli anni non possa essere minore di zero. Un vincolo è considerato una delle definizioni più importanti per molte metodologie moderne. Esiste infatti anche una sintassi in UML per specificare vincoli (se interessa, cfr. appendice L). Addirittura la sintassi UML stessa è definita tramite un linguaggio che si basa proprio sul concetto di vincolo. Tale linguaggio infatti si chiama Object Constraint Language (OCL).

I metadati sono quindi informazioni su informazioni. Nel nostro esempio la definizione di età era un metadato relativo alla definizione di persona. Ma relativamente alla definizione di età, potrebbe essere un metadato la definizione di anno. Quindi il concetto di metadato è sempre relativo a ciò che si sta definendo. Inoltre, un metadato è anche relativo all'interlocutore che ne ha bisogno. Se per esempio spieghiamo ad un adulto un concetto come quello di persona, non dovrebbe essere necessario specificare metadati. Se il nostro interlocutore è un bambino, invece, la situazione cambia.

Se poi il linguaggio in cui bisogna astrarre il concetto di persona è Java, allora non avremo di fronte un interlocutore umano, ma solo un freddo software. Che questo software sia il compilatore, la Java Virtual Machine, Javadoc o JAR non cambia molto. Questi software non possono capire se stiamo definendo correttamente la classe `Persona` a meno che non specifichiamo metadati.

Per le specifiche del linguaggio Java una classe è definita tecnicamente come “metamodello Java”.

Per esempio, definiamo la classe `Persona` nel seguente modo:

```
public class Persona {  
    // . . .  
    private int anni;  
    public void setAnni(int anni) {  
        if (anni < 0)  
            throw new IllegalArgumentException(anni + " anni, non è una età  
            valida!");  
        this.anni = anni;  
    }  
    public int getAnni() {  
        return anni;  
    }  
}
```

Con il controllo nel metodo `setAnni()` abbiamo specificato un vincolo destinato alla JVM. Un altro vincolo specificato dal codice precedente è invece riservato al compilatore: il modificatore `private`. Questo rappresenta un metadato il quale specifica al compilatore che la variabile `anni` non ha visibilità all'esterno della classe. Esistono in Java anche altri meccanismi per specificare metadati. Per esempio, potremmo specificare con il tag `@deprecated` interno ad un commento javadoc, che un certo metodo è deprecato. Questo tipo di metadato è invece destinato all'utility Javadoc e al compilatore. Su tag interni a commenti javadoc come `@deprecated` si basa una oramai deprecata tecnologia chiamata “Doclet”, la cui più famosa implementazione è un progetto open source rintracciabile all’indirizzo <http://xdoclet.sourceforge.net>. Un “motore doclet” è un software che crea file sorgenti Java in base a “tag” specificati nei commenti Javadoc con una determinata sintassi. Motori simili erano molto utilizzati dai tool di sviluppo più famosi per automatizzare la creazione di componenti complessi come gli Enterprise Java Beans (EJB). I tag doclet sono destinati al “motore doclet”, un software per la generazione di codice creato ad hoc.

Quindi Java fornisce diversi modi di specificare metadati, ma prima di Java 5 non esisteva una sintassi univoca per poter avere diversi interlocutori come il compilatore, Javadoc e la JVM. Inoltre, ognuno di questi modi (a parte la tecnologia doclet) è stato creato senza pensare al concetto di metadato. Alla fine ognuno di questi modi (compresa la tecnologia doclet) ha limiti palesi (che non elencheremo per semplicità) nello specificare i metadati. Giusto per fare un esempio, se consideriamo proprio **Xdoclet**, ecco come è possibile specificare un tag (possiamo anche ignorarne il significato semantico, concentriamoci sulla sintassi):

```
/** . . .  
 *  
 * @override . . .  
 **/
```

```
public void metodo() {}
```

il motore Xdoclet semplicemente ignorerà il tag, senza segnalare nessun tipo di errore (si notino le tre “r”). Infatti non ci possono essere controlli di correttezza visto che i tag sono definiti nell’area dove dovrebbero essere messi dei commenti. E nemmeno il compilatore Java si accorgerà di niente.

Le annotazioni invece possono avere il pieno supporto del compilatore Java, per cui il confronto termina qui.

Con l’introduzione dei tipi annotazioni o più semplicemente annotazioni (in inglese type annotations), Java ha un modo standard per definire i metadati di un programma. Per esempio è possibile specificare vincoli definiti dallo sviluppatore che possono essere interpretati da un software come la JVM, il compilatore, Javadoc o altro, ed è anche possibile creare un *processore di annotazioni* ad hoc che interpreta le annotazioni, per esempio creando nuovi file sorgenti ausiliari a quelli già esistenti. Ma le annotazioni, essendo meta-information dirette verso un software, hanno un orizzonte colmo di possibilità. Già abbiamo tanti riscontri sino a Java 8, ma siamo sicuri che il tempo ce ne porterà tanti altri.

In Java 8 è stato rimosso, insieme a tutte le relative librerie contenute nel package com.sun.mirror, lo strumento Annotation Processing Tool, meglio conosciuto come APT, presente invece nelle precedenti versioni.

12.1.1 Primo esempio

Tecnicamente le annotazioni sono un altro tipo di struttura dati del linguaggio, che si va ad aggiungere alle classi, alle interfacce e alle enumerazioni. Si dichiarano in maniera simile alle interfacce, ma si definiscono con una sintassi che va fuori dallo standard Java. Di seguito riportiamo un primo esempio di definizione di annotazione:

```
public @interface DaCompletere {  
    String descrizione();  
    String assegnataA() default "da assegnare";  
}
```

Come si può notare c’è una nuova parola chiave in Java: `@interface`. Questa serve proprio per definire un’annotazione come la parola chiave `class` serve a dichiarare una classe, o come `enum` serve a dichiarare un’enumerazione. In questo caso l’annotazione si chiama `DaCompletere` e definisce due “strani” metodi che sembrano astratti (ma non sono marcati dal modificatore `abstract`) : `descrizione()` e `assegnataA()`. Questi due metodi, detti anche elementi dell’enumerazione (in inglese annotation elements) hanno una sintassi abbreviata equivalente alla dichiarazione di una variabile con relativo metodo omonimo che restituisce il suo valore. Si noti anche come lo “strano” metodo `assegnataA()` utilizzi anche la parola chiave `default`. Questa serve a specificare un valore di `default` per il metodo nel caso che, utilizzando l’annotazione, lo sviluppatore non fornisca un valore per la variabile “invisibile” `assegnataA`. Ma approfondiremo tra poco i dettagli della sintassi.

DaCompletare viene detto quindi “tipo annotazione” (“annotation type”).

Dopo aver visto come si dichiarano le annotazioni cerchiamo di capire come si utilizzano. È possibile utilizzare un’annotazione come si fa con un modificatore, per esempio per un metodo, utilizzando la seguente sintassi:

```
public class Test {  
    @DaCompletare(  
        descrizione = "Bisogna fare qualcosa...",  
        assegnataA = "Cristina"  
    )  
    public void faQualcosa() {  
    }  
}
```

È anche possibile anteporre il modificatore (nell’esempio `public`) all’annotazione, non cambia nulla a parte la leggibilità. Si noti che `public` si riferirà sempre al metodo e non all’annotazione.

Analizziamo brevemente la sintassi utilizzata. L’istanza dell’annotazione viene dichiarata come se fosse un modificatore del metodo `faQualcosa()`. La sintassi però è molto particolare. Si utilizza il simbolo di chiocciola `@` (si dovrebbe leggere “AT”) che si antepone al nome dell’annotazione. Poi si aprono parentesi tonde per specificare i parametri, in modo simile a quanto si fa con un metodo. Specificare però i parametri di un’annotazione significa specificare coppie del tipo “chiave = valore”, dove le chiavi corrispondono alle variabili (mai dichiarate) relative al metodo omonimo. Quindi se nell’annotazione abbiamo definito i metodi `descrizione()` e `assegnataA()`, abbiamo anche in qualche modo definito le variabili “invisibili” (elementi) `descrizione` e `assegnataA`. Il loro tipo corrisponde esattamente al tipo di restituzione del metodo omonimo. Infatti tale metodo funzionerà da metodo getter (o accessor), ovvero restituirà il valore della relativa variabile.

Il metodo `faQualcosa()` è a questo punto stato annotato, ma manca ancora qualcosa. Bisogna creare un software che interpreti l’annotazione implementando un comportamento. Infatti, se anche possiamo intuire a cosa serva l’annotazione `DaCompletare`, nessun software avrà mai la nostra stessa perspicacia.

Per esempio, potremmo creare un’applicazione la quale riceva in input una classe, legga le eventuali annotazioni e pubblichi su una bacheca in Intranet i compiti da assegnare ai vari programmati.

```
import java.lang.reflect.*;  
import java.util.*;  
  
public class AnnotationsPublisher {  
    public static void main(String[] args) throws Exception {  
        Map<String, String> map = new HashMap<>();  
    }  
}
```

```

Method[] methods = Class.forName("Test").getMethods();
for (Method m: methods) {
    DaCompletere dc = null;
    SingleValue sv = null;
    if ((dc = m.getAnnotation (DaCompletere.class)) != null) {
        String descrizione = dc.descrizione();
        String assegnataA = dc.assegnataA();
        map.put(descrizione, assegnataA);
    }
}
publicaInIntranet(map);
}

public static void publicaInIntranet(Map <String, String>map) {
    Set <String>keys = map.keySet();
    for (String key: keys) {
        System.out.printf("Descrizione = %s; Assegnata a:%s", key,
                           map.get(key));
    }
}
}

```

La classe `AnnotationsPublisher` dichiara una mappa parametrizzata. Con un ciclo `foreach` estrae, tramite reflection, i metodi della classe `Test`. Sfruttando i nuovi metodi della classe `Method`, ed in particolare il metodo `getAnnotation()`, vengono scelti solo i metodi annotati con `DaCompletere`. Per essi vengono memorizzate nella mappa le informazioni. Finito il ciclo, viene invocato il metodo `pubblicaInIntranet()` che si occuperà di estrarre le informazioni dalla mappa per pubblicarle in Intranet.

Questo era solo un esempio, ma le potenzialità delle annotazioni sono teoricamente illimitate. Per semplicità abbiamo omesso la prima parte della definizione dell'annotazione `DaCompletere`. Senza questa parte non sarà possibile far funzionare correttamente l'esempio. Segue la definizione completa dell'annotazione:

```

import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletere {
    String descrizione();
    String assegnataA() default "da assegnare";
}

```

L'annotazione è stata a sua volta annotata, e tra poco capiremo perché. Ma se è possibile annotare un metodo ed un'annotazione, cos'altro possiamo annotare?

La lista è lunga: classi attributi, parametri, variabili locali... praticamente tutto! È possibile anche decidere se un'annotazione deve essere destinata alla lettura del solo compilatore o per essere utilizzata in fase di runtime. Di default l'annotazione viene riportata nel file `.class` ma non è considerata dalla JVM. Nei prossimi paragrafi chiariremo tutti gli aspetti delle annotazioni.

12.1.2 Tipologie di annotazioni e sintassi

Come per le enumerazioni, anche le annotazioni saranno alla fine trasformate in classi dal compilatore (necessario per la compatibilità con le versioni precedenti di Java). Le classi compilate implementeranno l'interfaccia `java.lang.annotation.Annotation`. Si noti che, implementando “manualmente” tale interfaccia, non definiremo un'annotazione. Praticamente tra le annotazioni e l'interfaccia `Annotation` c'è lo stesso rapporto vigente tra le enumerazioni e la classe `java.lang.Enum`.

Un'annotazione non può utilizzare la parola chiave `implements` né la parola chiave `extends`. Quindi in nessun modo un'annotazione può estendere un'altra annotazione, una classe, o implementare un'interfaccia.

Possiamo distinguere tre tipologie di annotazioni: le annotazioni ordinarie, le annotazioni a valore singolo e le annotazioni marcatorie.

12.1.2.1 Annotazione ordinaria (o completa)

L'unica annotazione che abbiamo già visto è un esempio di annotazione ordinaria. Si tratta del tipo di annotazione più complesso. Infatti viene detto anche annotazione completa (in inglese full annotation).

Rianalizziamo l'annotazione dell'esempio:

```
public @interface DaCompletere {  
    String descrizione();  
    String assegnataA() default "da assegnare";  
}
```

L'annotazione `DaCompletere` definisce due metodi astratti o che, almeno tecnicamente, hanno una sintassi simile a quella dei metodi astratti. Come già asserito in precedenza però, in realtà hanno un'implicita implementazione. Infatti, dichiarare un metodo equivale a dichiarare una coppia costituita da una variabile ed un metodo omonimi. Quest'ultimo restituisce il valore della variabile. Un'altra particolarità nella sintassi è l'utilizzo della parola chiave `default` per assegnare un valore predefinito alla variabile “nascosta” `assegnataA`. Possiamo quindi immaginare che un'annotazione sia un'oggetto di interfaccia che viene implementata da una classe creata al volo dal compilatore, simile alla seguente:

```
public class DaCompletereImpl implements DaCompletere {  
    private String descrizione;  
    private String assegnataA = "da assegnare";  
    public String descrizione() {  
        return descrizione;  
    }  
    public String assegnataA() {
```

```
        return assegnataA;
    }
}
```

Per un metodo di un'annotazione non è possibile né specificare parametri di input né indicare void come valore di ritorno. Il compilatore altrimenti segnalera errori esplicitamente.

Essendo come un'interfaccia, all'interno di un'annotazione è possibile dichiarare, oltre a metodi (implicitamente astratti) anche costanti ed enumerazioni (implicitamente public, static e final) e in generale tipi innestati (cfr. modulo 9). Per esempio possiamo arricchire l'annotazione in questo modo:

```
public @interface DaCompletere {
    String descrizione();
    String assegnataA() default "da assegnare";
    enum Priorita {ALTA, MEDIA, BASSA};
    Priorita priorita() default Priorita.ALTA;
}
```

La convenzione per gli identificatori delle annotazioni è identica a quella delle classi, delle interfacce e delle enumerazioni.

Dopo aver studiato la sintassi della dichiarazione di un'annotazione, analizziamo ora la sintassi dell'utilizzo di un'annotazione. Un'annotazione, come visto nell'esempio precedente, viene utilizzata come se fosse un modificatore. Per convenzione un'annotazione precede gli altri modificatori, ma non è obbligatorio. La sintassi per modificare un elemento di codice Java (classe, metodo, variabile locale, parametro etc.) è sempre del tipo:

```
@NomeAnnotazione ([lista di coppie] nome=valore)
```

Nel precedente esempio avevamo modificato un metodo nel seguente modo:

```
@DaCompletere(
    descrizione = "Bisogna fare qualcosa...",
    assegnataA = "Cristina"
)
public void faQualcosa() {
```

È obbligatorio impostare tutte le variabili i cui metodi non dichiarano un default. Quindi è legale scrivere:

```
@DaCompletere(  
    descrizione = "Bisogna fare qualcosa..."  
)  
public void faQualcosa() {  
}
```

mentre il seguente codice:

```
@DaCompletere(  
    assegnataA = "Cristina"  
)  
public void faQualcosa() {  
}
```

provocherà il seguente errore in compilazione:

```
Test.java:4: annotation DaCompletere is missing descrizione  
    assegnataA = "Cristina"  
                      ^  
1 error
```

Se invece consideriamo la versione arricchita dall'enumerazione dell'annotazione `DaCompletere`, allora potremmo utilizzarla nel seguente modo:

```
@DaCompletere(  
    descrizione = "Bisogna fare qualcosa...",  
    priorita = DaCompletere.Priorita.BASSA  
)  
public void faQualcosa() {  
}
```

Avendo fornito un default anche al metodo `priorita()` è possibile ometterne l'impostazione.

12.1.2.2 Annotazione a valore unico

Il secondo tipo di annotazione è detto annotazione a valore unico (in inglese single value annotation). Si tratta di un'annotazione contenente un unico metodo che viene chiamato `value()`. Per esempio, la seguente annotazione è di tipo a valore unico:

```
public @interface Serie {  
    Alfabeto value();
```

```
    enum Alfabeto {A,B,C};  
}
```

Si noti che è possibile dichiarare un qualsiasi tipo di restituzione per il metodo `value()` tranne il tipo `java.lang.Enum`.

Anche la seguente è un'annotazione a valore unico:

```
public @interface SingleValue {  
    int value();  
}
```

La seguente annotazione:

```
public @interface SerieOrdinaria {  
    Alfabeto alfabeto();  
    enum Alfabeto {A,B,C};  
}
```

è invece un'annotazione ordinaria, perché non definisce come unico elemento `value()`.

La differenza con altri tipi di annotazioni sta nella sintassi dell'utilizzo. Per utilizzare una annotazione a valore unico è infatti possibile scrivere all'interno delle parentesi dell'annotazione solamente il valore da assegnare. Per esempio, potremmo scrivere in luogo di:

```
@Serie(value = Serie.Alfabeto.A)  
public void faQualcosa() {  
}
```

più semplicemente:

```
@Serie(Serie.alfabeto.A)  
public void faQualcosa() {  
}
```

con lo stesso risultato.

12.1.2.3 Annotazione segnalibro

La terza tipologia di annotazioni è chiamata annotazione marcatrice (in inglese marker annotation). Si tratta della tipologia più semplice: un'annotazione che non ha metodi. Per esempio, il seguente è un esempio di annotazione segnalibro:

```
public @interface Marker {}
```

Questo tipo di annotazioni vengono utilizzate con la sintassi che ci si aspetterebbe:

```
@Marker()
public void faQualcosa() {
}
```

e possono risultare molto più utili di quanto non ci si aspetti.

È anche possibile evitare di specificare le parentesi tonde, come mostra il seguente codice:

```
@Marker public void faQualcosa() {
}
```

Si possono inoltre creare annotazioni innestate in classi, come nell'esempio seguente:

```
public class Test {
    public @interface Serie {
        Alfabeto value();
        enum Alfabeto {
            A,B,C
        };
    }
    @Serie(Serie.Alfabeto.A)
    public void faQualcosa() {
    }
}
```

Anche se in realtà è più probabile che si creino librerie pubbliche di annotazioni.

12.2 Annotare annotazioni (meta-annotazioni)

Nel package `java.lang.annotation` sono definite sei meta-annotazioni. Si chiamano `Retention`, `Target`, `Documented` e `Inherited`. A cui si aggiungono due nuove annotazioni introdotte con Java 8: `Native` e `Repeatable`. Tutte queste annotazioni servono solo ad annotare altre annotazioni.

In realtà abbiamo già visto un esempio di meta-annotazione nel paragrafo precedente. Infatti, dopo aver presentato il primo esempio, abbiamo evidenziato come l'annotazione `DaCompletare` per funzionare correttamente doveva essere annotata a sua volta dall'annotazione `Retention` nel seguente modo:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

In questo caso passando a `Retention` il valore unico `RetentionPolicy.RUNTIME`, abbiamo specificato che l'annotazione `DaCompletare` era destinata alla lettura da parte del runtime Java. Senza l'utilizzo della meta-annotazione, l'annotazione `DaCompletare` sarebbe stata inclusa nel bytecode di default dopo la compilazione, ma ignorata dal runtime Java. L'annotazione `Retention` è detagliata nel paragrafo 12.2.2.

12.2.1 Target

La prima meta-annotazione che affronteremo è `java.lang.annotation.Target`. Il suo scopo è specificare gli elementi del linguaggio a cui è applicabile l'annotazione che si sta definendo. Infatti, in italiano `Target` significa “obiettivo”. Questa meta-annotazione è di tipo *a valore singolo*, e prende come parametro un array di `java.lang.annotation.ElementType`. `ElementType` è un'enumerazione definita come segue:

```
package java.lang.annotation;
public enum ElementType {
    TYPE, // Classi, interfacce, o enumerazioni
    FIELD, // variabili d'istanza (anche se enum)
    METHOD, // Metodi
    PARAMETER, // Parametri di metodi
    CONSTRUCTOR, // Costruttori
    LOCAL_VARIABLE, // Variabili locali o clausola catch
    ANNOTATION_TYPE, // Tipi Annotazioni
    PACKAGE // Package
    TYPE_PARAMETER // Tipi parametro (visti con i Generics)
    TYPE_USE // Uso di un tipo
}
```

Essa specifica con i suoi elementi i vari elementi del linguaggio Java a cui è possibile applicare un'annotazione. Per esempio potremmo utilizzare questa meta-annotazione per limitare l'applicabilità dell'annotazione `DaCompletare`:

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType
@Target({TYPE, METHOD, CONSTRUCTOR, PACKAGE, ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

Ora l'annotazione `DaCompletare` è applicabile solo a classi, interfacce, enumerazioni, annotazioni, metodi, costruttori e package. Si noti come gli elementi dell'array di `ElementType` siano specificati mediante la sintassi breve dell'array (cfr. paragrafo 3.5.3). Inoltre tali elementi sono stati utilizzati senza essere referenziati grazie all'import statico. Notiamo inoltre che, se si vuole passare come parametro a `Target` un unico elemento, è possibile utilizzare anche la sintassi:

```
@Target(TYPE);
```

Il compilatore in questo caso è capace di capire le nostre intenzioni, e non ci obbliga ad utilizzare un array quando usiamo un solo elemento.

Se definiamo un'annotazione senza utilizzare la meta-annotazione Target, la nostra annotazione sarà applicabile di default a tutti gli elementi di tipo dichiarazione, tranne TYPE_PARAMETER e TYPE_USE.

Può risultare interessante notare come viene definita l'annotazione Target:

```
package java.lang.annotation;  
@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.ANNOTATION_TYPE)  
public @interface Target {  
    ElementType[] value();  
}
```

Quindi Target viene annotata anche da se stessa affinché sia applicabile solo ad altre annotazioni.

12.2.1.1 Type Annotations

Una novità importante che ha portato Java 8, è la possibilità di usare l'elemento dell'enumerazione ElementType.TYPE_USE. Prima dell'avvento di Java 8 era possibile annotare solo dichiarazioni, ora anche un qualsiasi uso di tipi (si parla di Type Annotations). Per esempio, ora è possibile annotare tipi quando si istanziano, nella clausola implements, nei cast, nella clausola throws e così via. Per esempio, supponendo che esista la seguente annotazione:

```
import java.lang.annotation.*;  
@Target(ElementType.TYPE_USE)  
@interface TestTP {  
}
```

Allora è possibile scrivere:

```
public class TypeParameterExample<@TestTP T> {  
    public void metodo (@TestTP T t){  
        System.out.println(t);  
    }  
}
```

In realtà per ora non esistono nella libreria ufficiale annotazioni con questo tipo di Target, ma l'Università di Washington ha rilasciato il suo Checker Framework, che ne fa ampio uso. Per

esempio in quest'ambizioso progetto è definita l'annotazione `@NotNull`, che può essere usata ogni volta che vogliamo che una certa variabile non debba diventare `null`. Questo framework è capace di interpretare il codice e capire come evitare la più frequente delle eccezioni al runtime: la `NullPointerException`. Indipendentemente dal fatto se userete o meno il Checker Framework, è importante notare quanto può essere potente utilizzare le type annotation. Nel caso dell'annotazione `@NotNull`, si sta cercando di eliminare uno dei più grandi problemi della programmazione Java, e questo è davvero notevole. Per chi fosse interessato al Checker Framework è possibile consultare il seguente link: <http://types.cs.washington.edu/checker-framework>.

Seguono alcuni esempi d'uso di type annotations all'interno di una classe molto complicata da leggere, ma che copre quasi tutti i casi di utilizzo:

```
import java.io.Serializable;
import java.util.*;
public class TPExamples<T extends @TestTP List<T>> extends @TestTP Object
    implements @TestTP Serializable {
    private static final long serialVersionUID = 1L;
    public static void main(String args[]) throws @TestTP Exception {
        byte i = (@TestTP byte)1000;
        Object o = new @TestTP String();
        if (o instanceof @TestTP String) {
            String s = (@TestTP String)o;
        }
        List<@TestTP String> list = null;
        List<? extends @TestTP Serializable> list2 = null;
        @TestTP String array @TestTP[] @TestTP[];
    }
}
```

Si può notare l'uso della type annotation `@TestTP` già nella dichiarazione della classe. La classe è generica ed ha un tipo parametro bounded il cui limite è annotato. Inoltre notare l'utilizzo dell'annotazione nella clausola `implements` e nella clausola `extends`. Anche il metodo `main()` è dichiarato con la clausola annotata `throws`. Poi l'annotazione viene usata in un cast di primitivi (variabile `i`) e in un cast di tipi complessi (variabile `s`), nonché all'interno del controllo con `instanceof`. Poi viene annotato anche un tipo parametro (della variabile `list`) ed un altro tipo parametro bounded wildcard. Infine un array bidimensionale viene annotato interamente (variabile `array[][]`) così come i suoi sotto-array (`array[i]` e `array[j]`, dove `i` e `j` rappresentano le righe e le colonne a cui solitamente pensiamo per convenzione). In definitiva si possono usare in quasi tutte le situazioni.

In particolare non si possono usare negli `import`. Per esempio è illegale scrivere:

```
import java.util.@TestTP List;
```

Così come i class literals. Ovvero, la seguente istruzione:

```
@TestTP List.class
```

non compilerà.

12.2.2 Retention

La meta-annotazione `Retention` (che in italiano possiamo tradurre come “conservazione”) è anch’essa molto importante. Serve per specificare come deve essere conservata dall’ambiente Java l’annotazione a cui viene applicata. Come `Target`, anche `Retention` è di tipo a singolo valore, ma prende come parametro un valore dell’enumerazione `java.lang.annotation.RetentionPolicy`. Segue la definizione di `RetentionPolicy` con commenti esplicativi sull’uso dei suoi valori:

```
package java.lang.annotation;
public enum RetentionPolicy {
    SOURCE, // l'annotazione è eliminata dal compilatore
    CLASS, /* l'annotazione viene conservata anche nel file
              ".class", ma ignorata dall JVM */
    RUNTIME /* l'annotazione viene conservata anche nel file
              ".class", e letta dalla JVM */
}
```

Come già affermato precedentemente la meta-annotazione `Retention` applicata alla annotazione `DaCompletare` farà in modo che le annotazioni di tipo `DaCompletare` siano conservate nei file compilati, per essere infine letti anche dalla virtual machine. Riportiamo nuovamente il codice di seguito:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

Nel paragrafo 12.1.1 denominato “Primo esempio” infatti, lo scopo di questa annotazione era essere letta tramite Reflection al runtime, e quindi era obbligatorio specificare la meta-annotazione.

12.2.3 Documented

Questa semplice meta-annotazione ha il compito di includere nella documentazione generata da Javadoc anche le annotazioni a cui è applicata. Per esempio, la meta-annotazione `Target` è a sua volta annotata da `Documented`, come mostra la sua dichiarazione:

```
package java.lang.annotation;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
```

```
public @interface Target {  
    ElementType[] value();  
}
```

Questo significa che, se generiamo la documentazione del nostro codice tramite il comando `javadoc` sui file dove è utilizzato `Target`, verrà riportata anche l'annotazione. Per esempio, se generiamo la documentazione della seguente annotazione:

```
import java.lang.annotation.*;  
@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.FIELD)  
public @interface MaxLength {  
    int value();  
}
```

che specifica la lunghezza massima di un certo campo (e che si può verificare a runtime) avremo come risultato quanto mostrato in Figura 12.1.

[Package](#) **Class** [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)
SUMMARY: [REQUIRED](#) | [OPTIONAL](#)

[FRAMES](#) [NO FRAMES](#)
DETAIL: [ELEMENT](#)

Annotation Type **MaxLength**

```
@Retention(value=RUNTIME)  
@Target(value=FIELD)  
public @interface MaxLength
```

Permette il controllo della lunghezza massima di un campo

Required Element Summary

int	value
-----	-----------------------

Figura 12.1 - Un particolare della documentazione generata.

In Figura 12.1 è possibile notare come anche `Retention` sia annotata con `Documented` e

quindi verrà documentata.

12.2.4 Inherited

Questa meta-annotazione permette alle annotazioni applicate a classi (e solo a classi) di essere ereditate. Ciò significa che se abbiamo la seguente annotazione:

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;
@Target({TYPE, METHOD, CONSTRUCTOR, PACKAGE, ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface DaCompletare {
    String descrizione();
    String assegnataA() default "da assegnare";
}
```

annotata a sua volta da `Inherited`, e l'applichiamo alla seguente classe:

```
@DaCompletare (
    descrizione = "Da descrivere..."
)
public class SuperClasse {
    //...
}
```

che a sua volta viene estesa dalla seguente sottoclasse:

```
public class SottoClasse extends SuperClasse {
    //...
}
```

Anche quest'ultima sarà annotata allo stesso modo della superclasse. Basterà eseguire la seguente classe per verificarlo:

```
import java.lang.reflect.*;
import java.util.*;
import java.lang.annotation.*;
public class AnnotationsReflection2 {
    public static void main(String[] args) throws Exception {
        Annotation[] dcs=SottoClasse.class.getAnnotations();
        for (Annotation dc: dcs) {
            System.out.println(dc);
        }
    }
}
```

Le annotazioni annotate con `Inherited` sono ereditate solo se applicate a classi. Tali annotazioni, se applicate a metodi, interfacce, o qualsiasi altro elemento Java che non sia una classe, non saranno ereditate.

Nel package `java.lang` sono definite le uniche tre annotazioni “normali” attualmente definite nella libreria: `Override`, `Deprecated` e `SuppressWarnings`.

12.2.5 Repeatable

A volte potrebbe essere necessario annotare più volte con la stessa annotazione un elemento. Per esempio, potremmo dichiarare un’annotazione chiamata `TestTrigger`, che potrebbe essere letta da un software esterno al fine di ottenere informazioni su come e quando fare il test di una certa classe. Per esempio:

```
@TestTrigger(quando="Ogni giorno, ore 18",
  come = TestTrigger.StrumentoDiTest.JUNIT)
@TestTrigger(quando="Ogni venerdì, ore 9",
  come = TestTrigger.StrumentoDiTest.GUI)
public class TestRepeatable {
  public void metodo(String... args) {
    // ...
  }
}
```

Abbiamo annotato con la stessa annotazione `TestTrigger` (che spiegheremo tra poco) la classe `TestRepeatable`. Prima di Java 8 questo tipo di sintassi avrebbe portato ad un errore in compilazione. Il programmatore avrebbe dovuto creare un’annotazione *contenitore* a mano, che permettesse di fatto di usare più volte l’annotazione. Ora invece, con l’introduzione della meta-annotazione `Repeatable`, è possibile dichiarare annotazioni che si possono applicare anche più di una volta.

Ma Java ha da sempre sposato il principio della compatibilità con le versioni precedenti. Se questo da un lato ha portato a creare qualche disagio con le nuove funzionalità che venivano introdotte (vedi Erasure per i generici introdotti in Java 5), ha di fatto permesso ai programmi di aggiornare la versione di Java senza grossi traumi. Anche in questo caso, per consentire le cosiddette repeating annotations (in italiano annotazioni che si ripetono) si è dovuto scendere a un compromesso: l’annotazione contenitore viene creata dal compilatore in base alle nostre direttive. Per creare una repeating annotation, bisogna per prima cosa annotare la nostra annotazione con `Repeatable` come nel seguente esempio:

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
@Repeatable(TestTriggers.class)
```

```

public @interface TestTrigger {
    String quando();
    StrumentoDiTest come() default StrumentoDiTest.JUNIT;
    public enum StrumentoDiTest {
        JUNIT, GUI, JMETER, SOAPUI;
    }
}

```

L'annotazione `TestTrigger` contiene una enumerazione innestata `StrumentoDiTest` che definisce i diversi modi di “come” deve essere eseguito il test. Inoltre definisce due elementi: `come` proprio di tipo `StrumentoDiTest`, e `quando` che invece è di tipo stringa. Questa annotazione `TestTrigger` è annotata da tutte le meta-annotazioni che abbiamo incontrato sinora, tra cui anche `Repeatable`. Quest’ultima è un’annotazione a valore unico che prende in input il tipo della annotazione contenitore (`TestTriggers`, notare il plurale) che il compilatore deve creare. Per fare ciò, dobbiamo almeno dichiarare l’annotazione che rappresenta il tipo contenitore, altrimenti il codice precedente non compila. Quindi creiamo l’annotazione che farà da contenitore, `TestTriggers`:

```

import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
public @interface TestTriggers {
    TestTrigger[] value();
}

```

Molto semplicemente deve essere un’annotazione a valore unico che ritorna un array di `TestTrigger`. È anche possibile leggere tramite Reflection le diverse annotazioni usate, usando il metodo `getAnnotationByType(Class<T>)` della classe `AnnotatedElement`, passandogli l’annotazione contenitore. Per esempio:

```

Class<TestRepeatable> testRepeatableClass = TestRepeatable.class;
TestTriggers [] annotations =
    testRepeatableClass.getAnnotationsByType(TestTriggers.class);
for (TestTriggers testTriggers: annotations) {
    TestTrigger[] values = testTriggers.value();
    for(TestTrigger testTrigger: values) {
        System.out.println(testTrigger.quando());
        System.out.println(testTrigger.come());
    }
}

```

In questo esempio abbiamo solo stampato i valori di `come()` e `quando()`, ma possiamo immaginare che sarebbe stato utile creare con una logica complessa dei metodi che interpretano questi valori, per poi schedularne i test per la classe annotata `TestRepeatable`. Per esempio:

```

public static void interpretaQuando(String quando) {

```

```
// codice complesso che interpreta la stringa
// e programma l'esecuzione del test
}
public static void interpretaCome(TestTrigger.SstrumentoDiTest come) {
    // codice complesso che registra lo strumento
    // di test per questo caso
}
```

Oppure è possibile usare il metodo `getAnnotations(Class<T>)` per farsi ritornare le occorrenze di utilizzo delle annotazioni tutte assieme. Consultare la documentazione della classe `AnnotatedElement` per tutti i metodi disponibili.

12.3 Annotazioni standard

Le prossime annotazione che vedremo, risiedono tutte nel package `java.lang`. Vengono dette annotazioni standard e sono molto utilizzate. Le annotazioni `Override`, `Deprecated` e `SuppressWarnings` sono state già nominate precedentemente tra le pagine di questo libro, e sono state le prime tre annotazioni create nella libreria Java. Invece le annotazioni `Native` e `FunctionalInterface` sono state introdotte con Java 8. Tutte comunque sono molto semplici da studiare e utilizzare.

12.3.1 `Override`

Come abbiamo già visto, l'annotazione `java.lang.Override` può essere utilizzata per indicare al compilatore che un metodo in una classe è un override di un altro metodo della sua superclasse. Di seguito è riportata la sua dichiarazione:

```
package java.lang;
import java.lang.annotation.*;
@Target(value=ElementType.METHOD)
@Retention(value=RetentionPolicy.SOURCE)
public @interface Override {}
```

Come è possibile notare si tratta di un'annotazione di tipo marcatrice, quindi non si devono specificare valori quando la si utilizza. Inoltre è applicabile solo a metodi (notare il valore della meta-annotazione `Target`). Infine, tramite il valore di `Retention` impostato a `RetentionPolicy.SOURCE`, viene specificato che `Override` è una annotazione interpretabile solo dal compilatore e che non sarà inserita nel bytecode relativo.

L'utilità di questa annotazione è piuttosto intuitiva. Per esempio, un tipico errore abbastanza difficile da scoprire per il programmatore Java, è quello di fare un override, ma sbagliare il nome del metodo nella sottoclassse. Usando questa annotazione dichiariamo al compilatore che stiamo consapevolmente facendo un override. Il compilatore quindi potrà eventualmente segnalaci un problema nel caso il nome del metodo non sia scritto correttamente. Inoltre aggiunge maggiore espressività al nostro codice: con un solo colpo d'occhio sapremo che il metodo che stiamo riscrivendo non è un metodo qualunque, ma uno riscritto da una superclasse.

Il baco del “nome sbagliato”, diventa ancora più difficile da risolvere quando sbagliamo un override estendendo una classe di un framework, dove il metodo riscritto deve essere chiamato dal framework stesso, e non da codice che scriviamo noi. Per esempio per scrivere un’applet (un’applicazione Java che gira all’interno di un browser, cfr. appendice Q) bisogna estendere la classe `java.applet.Applet` e riscriverne i metodi ereditati come `init()` e `start()`. Siccome in un’applet non esiste il metodo `main()`, questi metodi rappresentano il ciclo di vita dell’applicazione stessa. Quindi quando viene caricata l’applet viene prima chiamato il metodo `init()`, poi `start()` e così via. Basterebbe scrivere:

```
public void Init() {  
    // . . .  
}
```

(notare la lettera maiuscola) per perdere molto tempo a capire perché l’applet non viene inizializzata. Invece se annotiamo questo metodo con `@Override`, il problema viene evidenziato direttamente a livello di compilazione.

Come già asserito, il compilatore eseguito con l’opzione `-Xlint` segnalerà dei warning nel caso in cui trovi nel nostro codice `override` non annotati adeguatamente.

12.3.2 Deprecated

L’annotazione standard `java.lang.Deprecated` serve per indicare al compilatore e al runtime di Java che un metodo o un qualsiasi altro elemento di codice Java è deprecato. Ricordiamo che gli elementi in Java vengono deprecati allo scopo di segnalare agli utenti che lo stesso elemento non si dovrebbe più usare perché sostituito da qualche altro elemento nella sua funzionalità, e che in futuro potrebbe essere rimosso.

Quando si depreca un elemento è buona norma aggiungere anche un commento per specificare cosa sostituisce l’elemento deprecato.

Quindi l’annotazione `Deprecated` ha per il compilatore la stessa funzione che il tag `@deprecated` ha per l’utilità Javadoc. Sono due “istruzioni” complementari e vanno utilizzate contemporaneamente. Infatti se utilizzassimo solo il tag javadoc `@deprecated`, il compilatore ci restituirebbe un warning simile al seguente:

```
warning: [dep-ann] deprecated item is not annotated with @Deprecated
```

Anche il tag `Deprecated` è di tipo marker. Può essere utilizzato per annotare qualsiasi elemento Java. Segue la sua dichiarazione:

```
import java.lang.annotation.*;  
import static java.lang.annotation.RetentionPolicy.RUNTIME;
```

```
@Documented  
@Retention(value=RUNTIME)  
public @interface Deprecated {}
```

Il seguente codice rappresenta un esempio di utilizzo di `Deprecated`:

```
@DaCompletare (  
    descrizione = "Da descrivere ..."  
)  
public class SuperClasse {  
    /**  
     * Questo metodo è stato deprecato  
     * @deprecated utilizza un altro metodo per favore  
     */  
    @Deprecated public void metodo() {  
        . . .  
    }  
}
```

Le opzioni del compilatore `-deprecation` e `-Xlint:deprecation` sono equivalenti. Se viene per esempio utilizzato il metodo di `SuperClasse`, otterremo in fase di compilazione un warning simile al seguente:

```
TestAnnotation.java:4: warning: [deprecation] metodo() in SuperClasse has been  
deprecated  
    sc.metodo();  
    ^  
1 warning
```

sia che venga esplicitata l'opzione `-deprecation` sia che venga utilizzata l'opzione `-Xlint:deprecated`.

12.3.3 SuppressWarnings

Come abbiamo evidenziato più volte sino ad ora (anche con l'ultimo esempio), capiterà che a volte compilando il nostro codice verranno evidenziati dei warning dal compilatore. Per esempio, il seguente frammento di codice:

```
public static void main(String args[]) {  
    List strings = new ArrayList<String>();  
    strings.add("Lambda");  
    // . . .  
    Iterator<String> i = strings.iterator();  
    while (i.hasNext()) {  
        String string = i.next();  
        System.out.println(string);  
    }
```

```
}
```

avrà un esito di compilazione positivo, ma provocherà il seguente output:

```
Note: TestSuppressWarnings.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.
```

Tali warning (che nelle specifiche vengono definiti “lint”) avvertono lo sviluppatore che esistono operazioni non sicure o non controllate, e viene richiesto di ricompilare il file con l’opzione `-Xlint` per avere ulteriori dettagli. Seguendo il suggerimento del compilatore, ricompiliamo il file con l’opzione richiesta. L’output sarà:

```
D:\java8\Codice\modulo_12\esempi\TestSuppressWarnings.java:5: warning:  
[rawtypes] found raw type: List  
    List strings = new ArrayList<String>();  
           ^  
missing type arguments for generic class List<E>  
where E is a type-variable:  
    E extends Object declared in interface List  
D:\Codice\modulo_12\esempi\TestSuppressWarnings.java:6: warning:  
[unchecked] unchecked call to add(E) as a member of the raw type List  
    strings.add("Lambda");  
           ^  
where E is a type-variable:  
    E extends Object declared in interface List  
D:\java8\Codice\modulo_12\esempi\TestSuppressWarnings.java:10: warning:  
[unchecked] unchecked conversion  
    Iterator<String> i = strings.iterator();  
           ^  
required: Iterator<String>  
found: Iterator  
3 warnings
```

Come si può vedere sono stati segnalati tre warning. Il primo è di tipo “rawtypes”, visto che abbiamo utilizzato `List` senza specificare il tipo parametro `String`. Gli altri due sono di tipo “unchecked” in quanto la collezione `strings` viene aggiornata con il metodo `add()` senza che essa sia parametrizzata. Inoltre l’iteratore `i` viene usato con il tipo parametro `String` senza che il compilatore possa essere certo che la lista `strings` contenga stringhe, dato che non è stato dichiarato il tipo parametro.

Ignorare i warning solitamente non è saggio. Ma esistono casi in cui si preferisce lasciare il codice così com’è, magari perché è funzionante e non si vuole rischiare di creare bug. In tali casi è possibile utilizzare l’annotazione `SuppressWarnings`. Questa infatti può svolgere il ruolo di modificatore per classi, metodi, costruttori, variabili d’istanza, parametri e variabili locali, affinché non generino warning. Si tratta questa volta di un’annotazione a valore unico, il cui parametro è di tipo array di stringhe. Serve per specificare la tipologia di warning `Xlint` (per esempio `unchecked`) che deve

essere soppressa dal compilatore. Segue la sua dichiarazione:

```
package java.lang;
import java.lang.annotation.*;
import java.lang.annotation.ElementType;
import static java.lang.annotation.ElementType.*;
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

Per esempio, se vogliamo che il compilatore non generi warning né di tipo `unchecked` né di tipo `rawtypes` per il metodo dell'esempio precedente, basterà annotare tale metodo nel seguente modo:

```
@SuppressWarnings({"unchecked", "rawtypes"})
public static void main(String args[]) {
    List strings = new ArrayList<String>();
    strings.add("Lambda");
    // ...
    Iterator<String> i = strings.iterator();
    while (i.hasNext()) {
        String string = i.next();
        System.out.println(string);
    }
}
```

In particolare, tale annotazione consente di sopprimere warning riguardanti l'elemento annotato. Se l'elemento annotato contiene anche altri elementi che possono provocare warning (dello stesso tipo specificato) anche questi saranno soppressi dal compilatore. Per esempio, se una classe è annotata per sopprimere i warning di tipo `deprecated`, saranno soppressi eventuali altri warning dello stesso tipo relativi ai metodi della classe.

Come consiglio stilistico proveniente direttamente da Joshua Block (lo sviluppatore di tale annotazione), i programmati dovrebbero sempre annotare l'elemento più innestato. È sconsigliato annotare una classe per annotare tutti i suoi metodi. Sarebbe meglio annotare ogni metodo.

Se si specifica come parametro due volte la stessa stringa come valore (per esempio due volte `unchecked`), la seconda occorrenza sarà ignorata. Verranno anche ignorate tutte le occorrenze non valide per la sintassi Xlint.

12.3.4 FunctionalInterface

Questa annotazione è stata introdotta con Java 8. È dichiarata nel seguente modo:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

Si tratta di una interfaccia marcatrice ed è molto semplice intuire che il suo scopo sia quello di annotare una functional interface, ovvero una interfaccia che ha un unico metodo astratto (ma potrebbe avere tanti metodi statici o di default). Essendo annotata con `@Documented`, usare questa annotazione significa anche esplcitare sulla documentazione javadoc che l'interfaccia annotata è un'interfaccia funzionale. Non è obbligatorio usare questa annotazione affinché un'interfaccia sia considerata funzionale. Un'interfaccia si dice interfaccia funzionale se ha solamente un metodo che è astratto. Usando però quest'annotazione per l'interfaccia, il compilatore potrà segnalarci eventuali errori. Potrebbe capitare per esempio che una certa interfaccia sia dichiarata con un unico metodo astratto originariamente, ma che poi nel tempo qualcuno la vada a modificare aggiungendoci un altro metodo astratto. In casi come questo l'annotazione farà sì che il compilatore segnali un errore. Per esempio la compilazione della seguente interfaccia:

```
@FunctionalInterface
public interface FunctionalInterfaceExample {
    void metodoAstratto();
    void secondoMetodoAstratto();
    default void metodoDiDefault(){
        System.out.println("metodo di default");
    }
}
```

darà luogo al seguente errore:

```
D:\java8\Codice\modulo_12\esempi\FunctionalInterfaceExample.java:1:
error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
FunctionalInterfaceExample is not a functional interface
    multiple non-overriding abstract methods found in interface
FunctionalInterfaceExample
1 error
```

Come vedremo le interfacce funzionali sono molto importanti in Java, perché sono alla base della teoria delle espressioni lambda.

12.3.5 Native

Tra i vari modificatori che abbiamo incontrato sino ad ora, ce n'è uno di cui non abbiamo ancora parlato: il modificatore `native`. È stato deciso infatti di creare un'appendice apposita per la sua spiegazione in quanto è considerato un modificatore usato piuttosto raramente. Questo permette di far interagire codice Java con codice scritto in linguaggio nativo, ovvero in C/C++. Per maggiori dettagli consultare l'appendice M on line. L'annotazione `Native`, è così dichiarata:

```
@Documented  
@Target(ElementType.FIELD)  
@Retention(RetentionPolicy.SOURCE)  
public @interface Native {  
}
```

È quindi una annotazione marcatrice applicabile solo alla dichiarazione di attributi ed in particolare alle costanti. La sua funzione è quella di indicare che la costante annotata può essere referenziata da codice nativo. Un software scritto appositamente potrebbe generare file header (usati in C/C++) e in base a questa annotazione determinare se e come crearli.

Riepilogo

Dopo averne presentato la complessa sintassi, abbiamo visto come sia possibile creare proprie **annotazioni**. Abbiamo distinto tre tipologie di annotazioni: **marcatrici, a valore singolo e ordinarie**. Abbiamo anche visto cosa sono le meta-annotazioni (come per esempio `Target`) e come sia possibile creare annotazioni che annotano altre annotazioni. Infine abbiamo analizzato le cosiddette annotazioni standard della libreria, comprese quelle definite in Java 8.

Benché qualcuno potrebbe avere la sensazione che le annotazioni siano un argomento semplice, in realtà non lo sono affatto. Certo se ci si limita solo ad utilizzare le annotazioni standard sembra tutto molto banale. Ma in questo modulo abbiamo cercato di impostare il discorso in modo piuttosto tecnico, con lo scopo di dare un'idea delle potenzialità delle annotazioni. Per esempio, framework importanti come Java Enterprise Edition, Hibernate, Spring, Struts e JUnit si basano proprio su annotazioni create ad hoc per supportare alcune loro funzionalità.

Esercizi modulo 12

Esercizio 12.a) Annotazioni, dichiarazioni ed uso, Vero o Falso:

1. Un'annotazione è un modificatore.
2. Un'annotazione è un'interfaccia.
3. Gli elementi di un'annotazione sembrano metodi astratti ma sottintendono un'implementazione implicita.
4. La seguente è una dichiarazione di annotazione valida:

```
public @interface MiaAnnotazione {  
    void metodo();  
}
```

5. La seguente è una dichiarazione di annotazione valida:

```
public @interface MiaAnnotazione {  
    int metodo(int valore) default 5;  
}
```

- 6.** La seguente è una dichiarazione di annotazione valida:

```
public @interface MiaAnnotazione {  
    int metodo() default -99;  
    enum MiaEnum{VERO, FALSO};  
    MiaEnum miaEnum();  
}
```

- 7.** Supponendo che l'annotazione `MiaAnnotazione` definita nel punto 6 sia valida, nel seguente codice essa viene utilizzata correttamente:

```
public @MiaAnnotazione (  
    MiaAnnotazione.MiaEnum.VERO  
)  
MiaAnnotazione.MiaEnum m() {  
    return MiaAnnotazione.MiaEnum.VERO;  
}
```

- 8.** Supponendo che l'annotazione `MiaAnnotazione` definita nel punto 6 sia valida, nel seguente codice essa viene utilizzata correttamente:

```
public @MiaAnnotazione (  
    miaEnum=MiaAnnotazione.MiaEnum.VERO  
)  
MiaAnnotazione.MiaEnum m() {  
    return @MiaAnnotazione.miaEnum;  
}
```

- 9.** Consideriamo la seguente annotazione.

```
public @interface MiaAnnotazione {  
    int valore();  
}
```

Nel seguente codice essa viene utilizzata correttamente:

```
public @MiaAnnotazione (  
    5  
)  
void m()  
    //...  
}
```

- 10.** Consideriamo la seguente annotazione:

```
public @interface MiaAnnotazione {}
```

Nel seguente codice essa viene utilizzata correttamente:

```
public @MiaAnnotazione void m() {  
    //...  
}
```

Esercizio 12.b) Annotazioni e libreria, Vero o Falso:

- 1.** La seguente annotazione è anche una meta-annotazione:

```
public @interface MiaAnnotazione ()
```

2. La seguente annotazione è anche una meta-annotazione:

```
@Target (ElementType.SOURCE)
public @interface MiaAnnotazione ()
```

3. La seguente annotazione è anche una meta-annotazione:

```
@Target (ElementType.INTERFACE)
public @interface MiaAnnotazione ()
```

4. La seguente annotazione, se applicata ad un metodo, sarà documentata nella relativa documentazione Javadoc:

```
@Documented
@Target (ElementType.ANNOTATION_TYPE)
public @interface MiaAnnotazione ()
```

5. La seguente annotazione sarà ereditata se e solo se applicata ad una classe:

```
@Inherited
@Target (ElementType.METHOD)
public @interface MiaAnnotazione ()
```

6. Per la seguente annotazione è anche possibile creare un processore di annotazioni che riconosca al runtime il tipo di annotazione, per poter implementare un particolare comportamento:

```
@Documented
@Target (ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MiaAnnotazione ()
```

7. `Override` è un'annotazione standard per segnalare al runtime di Java che un metodo effettua l'`override` di un altro.

8. `Deprecated` in fondo può essere considerata anche una meta-annotazione perché applicabile ad altre annotazioni.

9. `SuppressWarnings` è una annotazione a valore singolo. `Deprecated` e `Override` invece sono entrambe annotazioni marcatrici.

10. Non è possibile utilizzare contemporaneamente le `SuppressWarnings`, `Deprecated` e `Override` su di un'unica classe.

Soluzioni esercizi modulo 12

Esercizio 12.a) Annotazioni, dichiarazioni ed uso, Vero o Falso:

1. **Falso**, è un tipo annotazione.

2. **Falso**, è un tipo annotazione.

3. Vero.

4. Falso, un elemento di un'annotazione non può avere come tipo `void`.

5. Falso, un metodo di un'annotazione non può avere parametri in input.

6. Vero.

7. Falso, infatti è legale sia il codice del metodo `m()` sia il dichiarare `public` prima dell'annotazione (ma qui è un modificatore del metodo). Non è legale però passare in input all'annotazione il valore `MiaAnnotazione.MiaEnum.VERO`, senza specificare una sintassi del tipo chiave = valore.

8. Falso, infatti la sintassi:

```
return @MiaAnnotazione.miaEnum;
```

non è valida. Non si può utilizzare un'annotazione come se fosse una classe con variabili statiche pubbliche.

9. Falso, infatti l'annotazione in questione non è a valore singolo, perché il suo unico elemento non si chiama `value()`.

10. Vero.

Esercizio 12.b) Annotazioni e libreria, Vero o Falso:

1. Vero, infatti se non si specifica con la meta-annotazione `Target` quali sono gli elementi a cui è applicabile l'annotazione in questione, l'annotazione sarà di default applicabile a qualsiasi elemento tranne tipi parametro e uso dei tipi.

2. Falso, il valore `ElementType.SOURCE` non esiste.

3. Falso, il valore `ElementType.@INTERFACE` non esiste.

4. Falso, non è neanche applicabile a metodi per via del valore di `Target`, che è `ElementType.ANNOTATION_TYPE`.

5. Falso, infatti non può essere applicata ad una classe se è annotata con `@Target (ElementType.METHOD)`.

6. Vero.

7. Falso, al compilatore, non al runtime.

8. Vero.

9. Vero.

10. Vero, `Override` non è applicabile a classi.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere cosa sono i metadati e la loro relatività (unità 12.1, 12.2)	<input type="checkbox"/>	
Comprendere l'utilità delle annotazioni (unità 12.1, 12.2, 12.3)	<input type="checkbox"/>	
Saper definire nuove annotazioni (unità 12.1, 12.2)	<input type="checkbox"/>	
Saper annotare elementi Java ed altre annotazioni (unità 12.2, 12.3)	<input type="checkbox"/>	
Saper utilizzare le annotazioni definite dalla libreria: le annotazioni standard e le meta-annotazioni (unità 12.3, 12.4)	<input type="checkbox"/>	
Comprendere e saper utilizzare le novità di Java 8 per le annotazioni come le repeating annotation, le type annotation e nuove annotazioni standard come FunctionalInterface (unità 12.2, 12.3)	<input type="checkbox"/>	

Note:

Le librerie di utilità: il package `java.util` e Date-Time API

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Saper implementare programmi configurabili mediante file di properties e preferences (unità 13.2.1).
- ✓ Saper implementare programmi con l'internazionalizzazione (unità 13.2.2, 13.2.3).
- ✓ Saper utilizzare la classe `StringTokenizer` per suddividere stringhe (unità 13.1.4).
- ✓ Saper utilizzare le basi delle espressioni regolari (unità 13.1.6).
- ✓ Comprendere le classi `Date` e `Calendar` (unità 13.1.7).
- ✓ Comprendere e saper utilizzare la nuova libreria “Date-Time API” (unità 13.2).

Questo modulo è dedicato alla parte della libreria standard con cui si ha a che fare ogni giorno, ovvero quella che definisce le sue classi di utilità. In particolare introdurremo le principali classi del package `java.util`, e dettaglieremo la nuova libreria nota come “Date-Time API” (“API” è l’acronimo di “Application Programming Interface” che potremmo tradurre semplicisticamente come “libreria”). Quest’ultima si pone l’obiettivo di sostituire il vecchio modo di gestire le date e gli orari con le storiche classi `Date`, `Calendar` e `GregorianCalendar`. Non vi è la pretesa di coprire tutte le circa 200 classi presenti in questi package, la documentazione ufficiale delle API di Java rimane sempre il riferimento principale. Cercheremo piuttosto di capire come utilizzare le classi e le interfacce principali. In particolare introdurremo il necessario per far interagire i nostri programmi con file di configurazione, internazionalizzare i propri contenuti, saper analizzare e formattare le stringhe, e manipolare date e orari.

13.1 Package `java.util`

Il package `java.util` è stato creato per fornire classi di utilità. Un buon esempio di classe di utilità è la classe `Objects`, che abbiamo incontrato nel paragrafo 11.1, e che offre metodi di utilità per oggetti generici. In questo package troverete anche altre classi equivalenti per array (la classe `Arrays`) e per le collection (classe `Collections`) che abbiamo anch’esse già incontrato nel paragrafo 11.1.9. Nello stesso paragrafo abbiamo anche studiato l’interfaccia funzionale `Comparator`, un’alternativa all’interfaccia `java.lang.Comparable` per ordinare gli elementi in array e collezioni. Ma `java.util` contiene anche classi per la gestione facilitata delle date e degli orari, classi per la gestione dell’internazionalizzazione e tante altre utilità come un analizzatore di stringhe, che vedremo in questo modulo. Inoltre sono definite classi di cui parleremo nei prossimi

moduli, come classi per gestire il modello ad eventi (cfr. “Observer” nell’appendice on line Q), classi per serializzare oggetti in Base64, scanner per leggere i caratteri digitati sulla tastiera (cfr. modulo 17 sull’IO), Timer e TimerTask per la creazione di timer (cfr. prossimo modulo sui Thread), un generatore di numeri casuali (cfr. classe Random), classi Optional per gestire eventuali valori indesiderati (cfr. modulo 16) e altro ancora. In questo package è soprattutto definita anche una serie di classi di utilità nota come il **framework Collections**, per gestire collezioni eterogenee di ogni tipo.

Con il termine “framework” in questo caso intendiamo un insieme di classi e di interfacce riutilizzabili ed estendibili. Questa definizione dovrebbe essere un po’ più complessa ma per i nostri scopi può andare bene.

Anche se abbiamo già usato più volte classi come ArrayList e HashMap, il framework Collections sarà affrontato in dettaglio nel modulo 16, insieme alla nuova “Stream API” e altre classi di utilità, alcune introdotte in Java 8 come StringJoiner. Quindi in questo modulo ci concentreremo sulle classi e le interfacce di uso comune.

13.1.1 Le classi Properties e Preferences

La classe Properties rappresenta un insieme di proprietà che possono essere salvate in un file. Estende la classe Hashtable (un’implementazione dell’interfaccia Map del framework Collections, molto simile ad HashMap introdotta nel paragrafo 10.1.5) e ne eredita i metodi. Rispetto ad un HashMap un oggetto Properties può leggere e salvare tutte le coppie chiave-valore in un file (detto file di properties) mediante i metodi load() e store(). Di fatto è possibile usare un file di properties come fosse un file dove memorizziamo le informazioni di configurazione della nostra applicazione (un file di configurazione). Come esempio prendiamo proprio i file sorgenti di EJE (disponibili per il download all’indirizzo <http://sourceforge.net/projects/eje>). Il file di properties chiamato EJE_options.properties viene utilizzato per leggere e salvare le preferenze dell’utente come la lingua, la versione di Java, la dimensione dei caratteri e così via. Per spiegare l’esempio ci concederemo un excursus su qualche classe del package java.io che poi approfondiremo nel modulo 17. Con il seguente codice (file EJE.java), tramite un blocco statico, viene chiamato il metodo loadProperties():

```
static {
    try {
        properties = new Properties();
        try {
            loadProperties();
            //...
        } catch (FileNotFoundException e) {
            //...
        }
    }
```

Consideriamo la variabile `properties` dichiarata come variabile d'istanza. Segue la dichiarazione del metodo `loadProperties()`:

```
public static void loadProperties() throws FileNotFoundException {
    InputStream is = null;
    try {
        is = new FileInputStream("resources/EJE_options.properties");
        properties.load(is);
    } catch (FileNotFoundException e) {
        //...
    }
}
```

L'oggetto `FileInputStream` che implementa la classe `InputStream` (e in questo esempio ne sfrutta anche un reference) permette di leggere il contenuto di un file. In questo caso al costruttore di `FileInputStream` abbiamo passato il file `EJE_options.properties` della cartella `resources`, un semplice file di testo al cui interno ci sono righe nel formato:

```
chiave=valore
```

Eccone un estratto:

```
eje.java.assertions=true
eje_area.braces_style=Java Style
eje.javac.target=1.8
file.recent8=D:\\\\java8\\\\Codice\\\\modulo_05\\\\esempi\\\\Punto.java
eje.javac.version=1.8
file.recent7=D:\\\\java8\\\\Codice\\\\modulo_07\\\\esempi\\\\Punto.java
eje_area.class_wizard=true
user.work_dir=D:\\\\java8\\\\Test;D:\\\\java8\\\\Codice;
file.recent6=D:\\\\java8\\\\Codice\\\\modulo_11\\\\esempi\\\\Punto.java
file.recent5=D:\\\\java8\\\\Codice\\\\modulo_11\\\\esempi\\\\PuntoHashCode.java
file.recent4=D:\\\\java8\\\\Codice\\\\modulo_12\\\\esempi\\\\
AnnotationsReAection2.java
file.recent3=D:\\\\java8\\\\Codice\\\\modulo_12\\\\esempi\\\\DaCompletare.java
file.recent2=D:\\\\java8\\\\Codice\\\\modulo_12\\\\esempi\\\\SottoClasse.java
file.recent1=D:\\\\java8\\\\Codice\\\\modulo_12\\\\esempi\\\\SuperClasse.java
eje_area.font.size=14
eje.java.docs=C:\\\\\\Program Files\\\\Java\\\\jdk1.8.0\\\\docs
eje_area.tab=4
eje.style=javax.swing.plaf.metal.MetalLookAndFeel
project.classpath=.;
eje.jdk.path=C:\\\\\\Program Files\\\\Java\\\\jdk1.8.0
project.output.path=
eje.lang=en
file.number.recent=8
project.docs.path=../docs
eje.java.version=1.8
```

```
eje_area.font.style=1  
eje_area.font.type=SansSerif
```

Chiamando il metodo `load()` sull'oggetto `properties`, quest'ultimo viene riempito sfruttando il formato delle sue righe. Quindi senza sforzi la nostra mappa sarà già pronta per l'uso. Per esempio per leggere la lingua usata da EJE, la proprietà con chiave `eje.lang` può essere letta con la seguente sintassi:

```
String language = properties.getProperty("eje.lang");
```

Nell'editor EJE, dopo che è stato aperto un file, il suo nome viene salvato nella lista dei file recenti. Viene impostato come primo file recente (dopo aver shiftato tutti gli altri di un posto). Per fare questo prima si imposta la proprietà nell'oggetto `properties`:

```
properties.setProperty("file.recent1", fileName);
```

dove la variabile `fileName` è il nome del file che è stato aperto in EJE. Poi viene chiamato il seguente metodo:

```
public static void saveProperties() {  
    OutputStream os = null;  
    try {  
        os = new FileOutputStream("resources/EJE_options.properties");  
        properties.store(os, "EJE OPTIONS - DO NOT EDIT");  
    } catch (FileNotFoundException e) {  
        . . .  
    }  
}
```

Come l'oggetto `FileInputStream` ci ha permesso di leggere il file, il `FileOutputStream` ci consente di scrivere all'interno del file. Con il metodo `store()` a cui passiamo l'oggetto `os` e una stringa che farà da testata (header) del file, salviamo nuovamente tutte le coppie “chiave=valore” riga per riga all'interno del file.

Il file di properties a volte viene salvato dopo averlo modificato “a mano”. Nel caso un valore di una certa chiave sia molto lungo, e lo si desideri riportare su più righe, bisogna utilizzare il simbolo di slash “/” alla fine di ogni riga quando si vuole andare a capo. Per esempio:

```
eje.message=This message is very very very very / very long
```

Gestire file di properties è fondamentalmente molto semplice. È anche possibile sfruttare i metodi `loadFromXml()` e `saveToXML()` per gestire semplici file di properties in formato XML. Il supporto di Java ad XML verrà approfondito nell'appendice N on line.

Una classe che offre funzionalità simili a `Properties` è la classe `Preferences` del package `java.util.prefs`. La differenza essenziale è che con la classe `Preferences` le coppie chiave-

valore non vengono inserite in un file specificato, bensì vengono memorizzate in un formato dipendente dalla piattaforma. Insomma se usiamo questa classe la nostra intenzione è modificarla solo attraverso l'applicazione e non "a mano".

L'utilizzo è abbastanza semplice e il seguente esempio dovrebbe testimoniarlo (leggere i commenti):

```
import java.util.prefs.Preferences;

public class TestPreferences {
    private Preferences preferences;
    private final static String key1 = "key1";
    private final static String key2 = "key2";
    private final static String key3 = "key3";

    public TestPreferences() {
        // Istanza di un oggetto Preferences.
        preferences = Preferences.userRoot();
    }

    public void putPreferences() {
        // Settiamo i valori con diversi tipi
        preferences.putBoolean(key1, false);
        preferences.put(key2, "Pluto");
        preferences.putInt(key3, 100);
    }

    public void printPreferences() {
        // Stampiamo il valore di key1, se non lo troviamo
        // stampiamo il default true.
        System.out.println(preferences.getBoolean(key1, true));
        // Stampiamo il valore di key2, se non lo troviamo
        // stampiamo il default pippo.
        System.out.println(preferences.get(key2, "pippo"));
        // Stampiamo il valore di key3, se non lo troviamo
        // stampiamo il default 0.
        System.out.println(preferences.getInt(key3, 0));
    }

    public void removePreferences() {
        // rimuoviamo il valore di key3
        preferences.remove(key3);
    }

    public static void main(String[] args) {
        TestPreferences test = new TestPreferences();
        test.putPreferences();
        test.printPreferences();
        test.removePreferences();
        test.printPreferences();
    }
}
```

}

In questa classe dichiariamo tre costanti statiche (`key1`, `key2`, `key3`) e un oggetto di tipo `preferences`. Nel costruttore di questa classe andiamo ad istanziare l'oggetto sfruttando il metodo statico `userRoot()` della classe `Preferences`. In particolare è possibile istanziare oggetti di tipo `Preferences` a più livelli, come nei nodi di un file XML. Infatti avremmo potuto anche ottenere un oggetto `Preferences` su un altro nodo con un'istruzione come la seguente:

```
preferences = Preferences.userRoot().node(this.getClass().getName());
```

In questo modo avremmo istanziato l'oggetto `preferences` su di un nodo figlio (a cui abbiamo passato il nome della classe `TestPreferences`), dove sarebbero state immagazzinate le coppie chiave-valore. La possibilità di poter definire diversi nodi è una delle principali differenze che ne caratterizzano la flessibilità rispetto agli oggetti `Properties`.

Il metodo `putPreferences()` imposta i valori delle coppie chiave-valore sfruttando oltre al classico metodo `put()` anche metodi basati sul tipo come `.putInt()` e `putBoolean()`. Il metodo `removePreferences()` invece rimuove la coppia la cui chiave era `key3`. Infine il metodo `printPreferences()` stampa i valori delle coppie dell'oggetto `preferences`. Si noti come i metodi “get” definiti dalla classe `Preferences` consentano di specificare anche un valore di default da ritornare nel caso la coppia richiesta non sia trovata.

L'output di questa classe sarà:

```
false  
Pluto  
100  
false  
Pluto  
0
```

Infatti, dopo la rimozione del valore di `key3`, viene stampato il valore 0. Per esercizio il lettore può modificare il metodo `main()` in modo tale da far stampare i valori di `key1`, `key2` e `key3` in una successiva esecuzione. In questo modo avrà anche la possibilità di testare che le variabili vengono conservate tra un'esecuzione e l'altra, il che testimonia che in qualche modo le variabili vengono memorizzate da qualche parte, senza necessità di un file di properties.

13.1.2 Classe Locale ed internazionalizzazione

Internazionalizzare un'applicazione significa che essa possa automaticamente adattarsi a vari linguaggi e zone, senza modificare il software.

In inglese il termine “internationalization” è spesso abbreviato in `i18n` poiché è un termine difficile da scrivere e pronunciare, e ci sono diciotto lettere tra la “i” iniziale e la “n” finale.

Per gestire l'internazionalizzazione non si può fare a meno di utilizzare la classe `Locale`. La classe `Locale` astrae il concetto di **zona**. Molte rappresentazioni di numerose altre classi Java dipendono da `Locale`. Per esempio per la rappresentazione di un numero decimale si utilizza la virgola come separatore tra il numero intero e le cifre decimali in Italia, mentre in America si utilizza il punto. Ecco che allora la classe `NumberFormat` (package `java.text`) può essere utilizzata nel seguente modo:

```
NumberFormat nf = NumberFormat.getInstance(Locale.ITALIAN);
Number number = nf.parse("25,1");
```

Nell'oggetto `number` sarà registrato il valore `25.1` dopo aver interpretato tramite il `Locale` italiano il numero con la virgola. Con il metodo `format()` invece è possibile formattare un numero con uno specifico `Locale`:

```
String s = nf.format(new Float(10.2));
System.out.println(s);
```

Altre entità che dipendono da `Locale` sono date, orari e valute (cfr. classe `Currency`).

La classe `Locale` è basata essenzialmente su tre variabili: `language`, `country` e `variant` (cfr. costruttori). Inoltre possiede alcuni metodi che restituiscono informazioni sulla zona. Ma probabilmente, più che interessarci dei metodi di `Locale`, ci dovremmo interessare all'utilizzo che ne fanno altre classi.

Infatti nella maggior parte dei casi non andremo ad istanziare `Locale`, bensì utilizzeremo le diverse costanti statiche di tipo `Locale` che individuano le zone considerate più facilmente utilizzabili. Esempi sono `Locale.US`, o `Locale.ITALY`. A volte potrebbe risultare utile il metodo `getDefault()`, che restituisce l'oggetto `Locale` prelevato dalle informazioni del sistema operativo.

Come esempio segue il codice con cui EJE va a riconoscere il `Locale` da utilizzare per impostare la lingua (cfr. prossimo paragrafo sul `ResourceBundle` per i dettagli):

```
static {
    ...
    Locale locale = null;
    String language = EJE.properties.getProperty("eje.lang");
    locale = (language != null && !language.equals("")) ?
        new Locale(language) : Locale.getDefault();
    ...
}
```

In questo caso viene prima letto dal file di properties il valore di `eje.lang`. Poi viene controllato che sia stato valorizzato in qualche modo. Se è stato valorizzato, mediante un operatore ternario, viene assegnata alla variabile `locale` un'istanza di `Locale` inizializzata con il valore della variabile `language`. Se `language` non è mai stato impostato (condizione che dovrebbe presentarsi appena scaricato EJE) la variabile `locale` viene impostata al suo valore di default.

filosofia dei plug-in, è stato introdotto un sottopackage di `java.util` chiamato `java.util.spi`. Questo contiene una serie di classi astratte che è possibile estendere per creare nuovi `Locale` in maniera conforme all'interfaccia fornita dal linguaggio. L'acronimo SPI sta per Service Provider Interface, e i sottopackage "spi" si trovano anche in altri package come `java.text`. L'installazione di un nuovo `Locale` si basa sul cosiddetto "Java Extension Mechanism", un procedimento standard per estendere le librerie già esistenti. Una breve descrizione è contenuta nella documentazione della classe `LocaleServiceProvider`, e non viene riportato in queste pagine perché non considerato fondamentale.

13.1.3 La classe ResourceBundle

La classe `ResourceBundle` rappresenta un contenitore di risorse dipendente da `Locale`. Per esempio EJE "parlerà" inglese o italiano sfruttando un semplice meccanismo basato su `ResourceBundle`. EJE utilizza al momento cinque file che si chiamano `EJE.properties`, `EJE_en.properties`, `EJE_es.properties`, `EJE_de.properties` e `EJE_it.properties` che come al solito contengono coppie del tipo chiave-valore. I primi due contengono tutte le stringhe personalizzabili in inglese (sono identici), il terzo le stesse stringhe in spagnolo, il quarto in tedesco e il quinto in italiano. Con l'ultimo esempio di codice, abbiamo visto come viene scelto il `Locale`. Attualmente ci sono cinque possibilità per EJE:

1. il `Locale` è esplicitamente italiano perché la variabile `language` vale "it";
2. il `Locale` è inglese perché la variabile `language` vale "en";
3. il `Locale` è spagnolo perché la variabile `language` vale "es";
4. il `Locale` è tedesco perché la variabile `language` vale "de";
5. il `Locale` è quello di default (quindi potrebbe essere anche francese) perché la variabile `language` non è stata valorizzata.

Per EJE il default è comunque inglese. Infatti i file `EJE.properties` ed `EJE_en.properties` sono identici.

Il codice che imposta il `ResourceBundle` (di nome `resources`) da utilizzare è il seguente:

```
static {  
    try {  
        // . . .  
        Locale locale = null;  
        String language = EJE.properties.getProperty("eje.lang");  
        locale = (language != null && !language.equals("")) ?  
            new Locale(language) : Locale.getDefault();  
        // . . .  
        resources = ResourceBundle.getBundle("resources.EJE", locale);  
    } catch (Exception e) {  
        // . . .  
    }  
}
```

```
    } catch (MissingResourceException mre) {
        // ...
    }
}
```

dove la variabile `locale` è stata impostata come abbiamo visto nell'esempio precedente. Con il metodo `getBundle()` stiamo chiedendo di caricare un file che si chiama `EJE` nella directory `resources`, con il `locale` specificato. In modo automatico, se il `locale` specificato è italiano, viene caricato il file `EJE_it.properties`; se il `locale` è inglese viene caricato il file `EJE_en.properties` etc. Se il `locale` è per esempio francese, `EJE` prova a caricare un file di nome `EJE_fr.properties` che però non esiste, quindi viene caricato il file di default `EJE.properties`. Tutto sommato il codice scritto è minimo.

Nel resto dell'applicazione, quando bisogna visualizzare una stringa internazionalizzata, viene semplicemente chiesto di farlo al `ResourceBundle` mediante il metodo `getString()`. Per esempio nel seguente modo:

```
newMenuItem = new JMenuItem(resources.getString("file.new"),
    new ImageIcon("resources" + File.separator + "images" +
    File.separator + "new.png"));
```

la voce di menu relativa al comando “nuovo file” carica la sua stringa direttamente dal `ResourceBundle resources`.

Se il lettore è interessato a scrivere un file di properties per EJE diverso da italiano ed inglese (per esempio francese, russo, portoghese etc.) può contattare l'autore all'indirizzo `eje@claudiodesio.com`. Sarà poi citato nella documentazione in linea del software come contributor.

La classe innestata statica `ResourceBundle.Control` serve a dare alle applicazioni maggior controllo sul processo di caricamento di un `ResourceBundle`. Questa classe infatti definisce una serie di metodi di callback (che vengono chiamati dalla JVM in base a determinati eventi) che vengono invocati dal metodo `ResourceBundle.getBundle()` durante il caricamento del bundle. È quindi possibile fare override di questi metodi per gestire il comportamento del metodo `getBundle()`.

13.1.4 La classe StringTokenizer

`StringTokenizer` è una semplice classe che permette di separare i contenuti di una stringa in più parti (“token”). Solitamente si utilizza per estrarre le parole di una stringa. I costruttori vogliono in input una stringa e consentono di estrarne i token. Ma un oggetto di tipo `StringTokenizer` ha bisogno di sapere anche come identificare i token. Per questo si possono esplicitare i delimitatori dei token. Un token è quindi, in generale, la sequenza massima di caratteri consecutivi che non sono delimitatori. Per esempio:

```
 StringTokenizer st = new StringTokenizer("questo è un test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

stampa il seguente output:

```
questo
è
un
test
```

Questo perché con il costruttore non abbiamo specificato i delimitatori, che per default saranno i seguenti: “\t\n\r\f” (si noti che il primo delimitatore è uno “spazio”). Se utilizzassimo il seguente costruttore:

```
 StringTokenizer st = new StringTokenizer("questo è un test", " ");
```

l’output sarebbe costituito da un unico token dato che non esiste quel delimitatore nella stringa specificata:

```
questo è un test
```

Esiste anche un terzo costruttore che riceve in input un booleano. Se questo booleano vale `true`, lo `StringTokenizer` considererà token anche gli stessi delimitatori. Quindi l’output del seguente codice:

```
 StringTokenizer st = new StringTokenizer("questo è un test", "t", true);
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

sarà:

```
ques
t
o è un
t
es
t
```

mentre se utilizzassimo questo costruttore:

```
 StringTokenizer st = new StringTokenizer("questo è un test", "t", false);
```

l’output sarebbe:

ques
o è un
es

13.1.5 Formattazioni di output

Uno degli esempi più riusciti di utilizzo dei varargs è sicuramente il metodo `format()` della classe `java.util.Formatter`. Il metodo ha anche un overload che consente di specificare un `Locale` per effettuare eventuali formattazioni basate sull'internazionalizzazione. Seguono le dichiarazioni del metodo `format()`:

```
public Formatter format(String format, Object... args)  
public Formatter format(Locale l, String format, Object... args)
```

Il metodo `format()` serve per stampare in output, con la possibilità di formattare correttamente tutti gli input che gli vengono passati. Consideriamo il seguente esempio:

```
Formatter formatter = new Formatter(System.out);  
formatter.format(Locale.ITALY, "e = %+10.4f", Math.E);
```

Esso stamperà il seguente output:

```
e = +2,7183
```

Con la sintassi:

```
%+10.4f
```

abbiamo specificato secondo la sintassi di formattazione definita dalla classe `Formatter` il valore `double` della costante `E` della classe `Math`. In particolare, con il simbolo “%” avvertiamo il formatter che stiamo definendo un valore da formattare, in questo caso il primo (e l’unico) valore del parametro varargs, ovvero `Math.E`. Con il simbolo “+” invece, abbiamo specificato che l’output deve obbligatoriamente specificare il segno (positivo o negativo che sia). La “f” finale serve per specificare che l’output deve essere formattato come numero decimale (floating point). Il numero “10” che precede il “.” fa in modo che l’output sia formattato in almeno dieci posizioni. Se l’output è costituito da meno di dieci posizioni come nel nostro caso, il valore viene emesso in dieci caratteri allineato a destra, come è possibile notare dall’esempio. Infine, il numero “4” specificato dopo il “.” indica che il valore deve essere emesso con quattro decimali (con eventuale arrotondamento). La classe offre la possibilità di formattare gli output con un numero enorme di varianti (cfr. documentazione classe `Formatter`). L’argomento sembra piuttosto complesso ma esiste una categoria di persone che non sarebbe d’accordo con questa affermazione: i programmati C/C++. Essi infatti hanno dovuto imparare ben presto le regole che governano la formattazione degli output. Infatti, la principale funzione di stampa del C si chiama `printf()` e formatta gli output proprio con le regole a cui abbiamo appena accennato. Ecco che allora ci ritroviamo nella classe `PrintStream` un

metodo chiamato proprio `printf()`. Questo ricalca la firma del metodo `format()` di `Formatter`, sfruttando come secondo parametro un varargs di `Object`. Infatti `printf()` non fa altro che chiamare a sua volta il metodo `format()`. Ma qual è l'oggetto di tipo `PrintStream` più famoso? `System.out` naturalmente! Da ora in poi sarà possibile utilizzare la seguente sintassi per stampare:

```
System.out.printf("Data %d", new Date());
```

Quindi siamo di fronte ad un modo più familiare (per i programmatore che vengono dal C) di utilizzare il metodo `format()`. Come al solito il lettore può conoscere ogni dettaglio consultando la documentazione ufficiale.

Se come parametro varargs del metodo `format()` viene passato un array di `Object` ci troveremo davanti ad una situazione imprevista. Consideriamo il seguente esempio:

```
Object [] o = {"Java 1.5", "Java 5", "Tiger"};
System.out.printf("%s", o);
```

L'output del precedente codice sarà:

```
Java 1.5
```

Infatti il formatter considererà parte del parametro varargs tutti i singoli elementi dell'array `o`. Per far sì che l'output sia formattato come previsto e che l'array di `Object` sia considerato come singolo oggetto, è possibile utilizzare la seguente sintassi:

```
System.out.printf("%s", new Object[]{ o });
```

o equivalentemente:

```
System.out.printf("%s", (Object)o);
```

13.1.6 Espressioni regolari

Esiste un modo però molto più potente di analizzare testo in Java. Infatti dalla versione 1.4 Java supporta le cosiddette **espressioni regolari** (in inglese **regular expressions**). Si tratta di una potente tecnica che esiste anche in altri ambienti (per esempio i sistemi Unix e il linguaggio Perl), per la ricerca e l'analisi del testo. Le espressioni regolari si basano su di un linguaggio sintetico e a volte criptico, composto appunto di espressioni, con una sintassi non ambigua, per individuare determinate aree di testo. Per esempio, l'espressione:

```
[aeiou]
```

permette di individuare una vocale. La sintassi è abbastanza vasta e quindi ci limiteremo a riportare solo quella fondamentale.

Essenzialmente ci occorre conoscere tre tipologie di espressioni: i **gruppi di caratteri (character**

classes), le classi predefinite e i quantificatori (greedy quantifier). Un esempio di **gruppi di caratteri** l'abbiamo appena visto:

```
[aeiou]
```

che individua una tra le vocali. Per esempio l'espressione:

```
alunn[ao]
```

consente di individuare all'interno di un testo le occorrenze sia della parola “alunno” che della parola “alunna”. L'operatore ^ è detto **operatore di negazione**, e per esempio l'espressione:

```
[^aeiou]
```

individua un qualsiasi carattere escluse le vocali (quindi una consonante). Con i gruppi di caratteri è anche possibile specificare range di caratteri con il separatore -. Per esempio l'espressione:

```
[a-zA-Z]
```

permette di individuare una qualsiasi lettera maiuscola o minuscola.

Le **classi predefinite** sono particolari classi che consentono di definire alcune espressioni con una sintassi abbreviata e quindi semplificata. Segue uno schema con tutte le classi predefinite e le relative descrizioni:

.	Un qualsiasi carattere
\d	Una cifra (equivalente a [0-9])
\D	Una non-cifra (equivalente a [^0-9])
\s	Un carattere spazio bianco (equivalente a [\t\n\x0B\f\r])
\S	Un carattere non-spazio bianco (equivalente a [^\s])
\w	Un carattere (equivalente a [a-zA-Z_0-9])
\W	Un non carattere (equivalente a [^\w])

I **quantificatori** servono a indicare la molteplicità di caratteri, parole, espressioni o gruppi di caratteri. In particolare:

- * significa zero o più occorrenze del pattern precedente;
- + significa una o più occorrenze del pattern precedente;
- ? significa zero o una occorrenza del pattern precedente.

In realtà stiamo parlando solo di una tipologia di quantificatori (i greedy quantifier) ma ne esistono altre con potenzialità notevoli.

La libreria Java supporta le espressioni regolari con tante classi: per esempio il metodo `replace()` della classe `String`, oppure la classe `Scanner` che introdurremo nel modulo 17 riguardante l'Input-Output (la quale potrebbe tranquillamente sostituirsi anche alla classe `StringTokenizer`, ma usando espressioni regolari). Esiste il package `java.util.regex` che definisce due semplici classi sulla quali si basa tutta la libreria che utilizza le espressioni regolari. Si tratta delle classi `Pattern` e `Matcher`. La classe `Pattern` serve proprio a definire le espressioni regolari mediante il suo metodo statico `compile()`. La classe `Matcher` invece definisce diversi metodi per la ricerca e l'analisi del testo, come i metodi `matches()`, `find()`, `start()`, `end()`, `replaceFirst()`, `replaceAll()` etc. EJE fa grande uso di espressioni regolari per interpretare il codice digitato dallo sviluppatore. Il seguente codice:

```
Pattern p = Pattern.compile("\bpackage\b");
String content = EJEArea.this.getText();
Matcher m = p.matcher(content);
while (m.find()) {
    int start = m.start();
    int end = m.end();
    . .
}
```

permette di ricercare la posizione della parola “package” all’interno del testo eseguendo un ciclo su ogni occorrenza trovata. Si tenga presente che la variabile `content` contiene il testo digitato dall’utente su EJE, e che il simbolo `\b` rappresenta il delimitatore dell’espressione.

Si noti come sia stato necessario utilizzare due simboli di backslash invece che uno solo. Infatti il primo è necessario alla sintassi Java per interpretare il secondo come simbolo di backslash!

13.1.7 Date, orari e valute

Le classi `Date`, `Calendar`, `GregorianCalendar`, `TimeZone` e `SimpleTimeZone` permettono di gestire ogni tipologia di data e orario. È possibile anche gestire tali entità sfruttando l’internazionalizzazione. Non sempre sarà semplice maneggiare date e orari, e questo è dovuto proprio alla complessità delle entità (a tal proposito basta leggere l’introduzione della classe `Date` nella documentazione ufficiale). La complessità è aumentata dal fatto che, oltre a queste classi, è molto probabile che serva utilizzarne altre quali `DateFormat` e `SimpleDateFormat` (package `java.text`) le quali consentono la trasformazione da stringa a data e viceversa. Vista la vastità del discorso, preferiamo solo accennare qualche esempio di utilizzo comune di tali classi. Per il resto si rimanda direttamente alla documentazione ufficiale. Soprattutto nel prossimo paragrafo sarà introdotta la libreria destinata a rimodernare il modo in cui si gestiscono queste entità. Tuttavia le classi spiegate in questo paragrafo sono comunque importanti visto che tutto il codice Java scritto sino all’avvento di Java 8 ne ha fatto uso. Nel paragrafo 13.2.5 vedremo anche come è possibile migrare dal vecchio a nuovo modo di gestire le date. Partiamo quindi direttamente con degli esempi.

Il seguente codice crea un oggetto `Date` (contenente la data corrente) e ne stampa il contenuto, formattandolo mediante un oggetto `SimpleDateFormat` creato col pattern “dd-MM-yyyy”, ovvero due

cifre per il giorno, due per il mese, e quattro per l'anno:

```
Date date = new Date();
SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy");
System.out.println(df.format(date));
```

I due seguenti frammenti di codice, invece, producono esattamente lo stesso output di formattazione di tempi. Nel primo caso viene utilizzato un pattern personalizzato:

```
DateFormat formatter = new SimpleDateFormat("HH:mm:ss");
String s = formatter.format(new Date());
s = DateFormat.getTimeInstance(DateFormat.MEDIUM, locale).format(new
Date());
System.out.println(s);
```

Nel seguente codice invece raggiungiamo lo stesso risultato chiedendo di formattare allo stesso modo in maniera breve (`DateFormat.SHORT`) secondo lo stile Italiano:

```
Locale locale = Locale.ITALY;
DateFormat formatter =
DateFormat.getTimeInstance(DateFormat.SHORT, locale);
String s = formatter.format(new Date());
s = DateFormat.getTimeInstance(DateFormat.MEDIUM, locale).format(new
Date());
System.out.println(s);
```

Nel prossimo esempio formattiamo due valute (o numeri) secondo gli standard americano ed italiano:

```
double number = 55667788.12345;
Locale localeUsa = Locale.US;
Locale localeIta = Locale.ITALY;

NumberFormat usaFormat = NumberFormat.getInstance(localeUsa);
String usaNumber = usaFormat.format(number);
System.out.println(localeUsa.getDisplayCountry() + " " + usaNumber);

NumberFormat itaFormat = NumberFormat.getInstance(localeIta);
String itaNumber = itaFormat.format(number);
System.out.println(localeIta.getDisplayCountry() + " " + itaNumber);
```

di cui l'output è:

```
Stati Uniti 55,667,788.123
Italia 55.667.788,123
```

Si noti come l'oggetto `Locale` può essere utile per certe formattazioni.

Nel prossimo esempio utiliziamo una classe molto importante, non dal punto di vista della

formattazione delle date, ma proprio per il calcolo: `Calendar` (e la sua sottoclasse `GregorianCalendar`). Se non trovate i metodi che vi servono nella classe `Date`, date un occhiata alla documentazione di queste classi, potrete risolvere i vostri dubbi. Per esempio, il metodo `get()`, utilizzando le costanti di `Calendar`, consente di recuperare le singole parti di una data:

```
public int getNumeroSettimana(Date date) {  
    Calendar calendar = new GregorianCalendar();  
    calendar.setTime(date);  
    int settimana = calendar.get(Calendar.WEEK_OF_YEAR);  
    return settimana;  
}
```

Esistono tante altre costanti statiche di `Calendar`: `DAY`, `MONTH`, `YEAR`, `DAY_OF_WEEK`, `DAY_OF_YEAR`, `HOUR` etc.

La sottoclasse `GregorianCalendar` possiede anche un costruttore che prende come argomento un oggetto di tipo `TimeZone`, con il quale Java gestisce il fuso orario. Con questo è possibile ottenere orari relativi ad altre parti del mondo. Per esempio:

```
Calendar cal = new  
GregorianCalendar(TimeZone.getTimeZone("America/Denver"));  
int hourOfDay = cal.get(Calendar.HOUR_OF_DAY);
```

permette di vedere l'ora attuale a Denver negli Stati Uniti. Per ottenere tutti gli id validi per il metodo `getTimeZone()`, è possibile invocare il metodo statico `getAvailableIDs()` della stessa classe `TimeZone` che restituisce un array di stringhe.

13.2 La novità di Java 8: Date-Time API

Gestire date e orari in Java non è mai stato semplice. Se da un lato c'è stato uno sforzo notevole nel progettare questa libreria in modo tale da essere completa ed efficiente, da un altro si può dire che la complessità sia mediamente alta. Inoltre casi particolari non sono gestiti. Per esempio in alcune nazioni il passaggio dall'ora solare alla legale e viceversa è fatto più di una volta all'anno, in altre ci sono degli anni dove il passaggio non viene proprio fatto. Altra difficoltà sorgono quando si utilizzano fusi orari, anni bisestili o anni antecedenti al 1970. Infine lo standard su cui si basano le classi `Date` e `Calendar` soffrono di piccole imprecisioni di calcolo temporale basandosi sui tempi della macchina su cui girano. I progettisti di Java 8 quindi, spinti dalla comunità Java, hanno deciso di implementare una nuova libreria nota come “Date-Time API”. Questa libreria contiene solo classi immutabili, ovvero gli oggetti istanziati da esse non possono più essere modificati. Dovrebbe inoltre caratterizzarsi dalla possibilità di essere una libreria chiara, semplice da utilizzare ed estensibile. Questa API non si trova nel package `java.util`, bensì nel package `java.time` e suoi sottopackage. In particolare:

- ❑ `java.time`: contiene le classi fondamentali della libreria, che rappresentano tempi, date, periodi, istanti, etc. basati sullo standard ISO-8601.
- ❑ `java.time.chrono`: contiene le classi equivalenti a quelle di `java.time`, ma che seguono

standard diversi da quelli di ISO-8601. È anche possibile creare nuovi calendari personalizzati.

- ❑ `java.time.zone`: contiene le classi per gestire i fusi orari come `ZoneDateTime`, `ZoneOffset` e `ZoneId`.
- ❑ `java.time.temporal`: package dedicato essenzialmente a chi deve estendere le funzionalità della libreria. Ma qui sono anche definiti i *fields* e gli *unit* di cui parleremo più avanti.
- ❑ `java.time.format`: contiene le classi per formattare e interpretare le date e gli orari.

13.2.1 Standardizzazione dei metodi

Per renderne semplice l'utilizzo, la maggior parte dei metodi contenuti in questi package rispettano degli standard per i nomi. Per esempio in diverse classi esiste un metodo statico `from()` che permette di istanziare un tipo della classe su cui si chiama il metodo, partendo dall'oggetto specificato come parametro del metodo. Per esempio il seguente frammento di codice:

```
LocalDate localDate = LocalDate.now();
YearMonth ym = YearMonth.from(localDate);
```

istanzia un oggetto `LocalDate` (che rappresenta una data, come vedremo tra poco) tramite il metodo `now()`, che restituisce la data contestuale al momento in cui si esegue il programma. Poi il metodo statico `from()` della classe `YearMonth` restituisce un nuovo oggetto `YearMonth` a partire dall'oggetto `localDate`.

La seguente tabella riporta una serie di identificatori (o prefissi di identificatori) di metodi, utilizzati in molte classi della libreria “Date-Time API”. Si noti che non esistono metodi di tipo “set”, visto che tutte le istanze della libreria sono immutabili. Ogni metodo che intende modificare un oggetto in essere, ne restituisce uno nuovo dello stesso tipo (un po’ come avviene con i metodi della classe `String` o delle classi wrapper):

Nome Prefisso	Static	Descrizione	Esempio
now	Sì	Restituisce un'istanza della classe su cui è chiamato relativa all'istante in cui è invocato il metodo.	<pre>LocalDate ora = LocalDate.now();</pre>
from	Sì	Restituisce un'istanza della classe su cui è chiamato partendo dal parametro <code>in input</code> che deve essere più completo (con più informazioni). Questo implica perdere qualche informazione rispetto all'oggetto passato in <code>input</code> .	<pre>// localDate contiene // l'informazione del // giorno in più rispetto a YearMonth YearMonth ym = YearMonth. from(localDate);</pre>
of	Sì	Restituisce un'istanza della classe su cui è chiamato dopo aver validato l' <code>input</code> .	<pre>LocalDate fulvioBirth = LocalDate.of(1974, 9, 5); fulvioBirth = LocalDate. of(1974, Month.SEPTEMBER, 5);</pre>
with	No	Restituisce una copia dell'oggetto passato in <code>input</code> con un elemento cambiato. È il metodo equivalente ad un metodo "set", per un oggetto immutabile.	<pre>dateTimeFormatter = dateTimeFormatter. withLocale(Locale.FRENCH);</pre>
plus	No	Restituisce una copia del parametro <code>in input</code> con l'aggiunta del tempo specificato.	<pre>Instant traDieciMinuti = Instant.now().plus(10, ChronoUnit.MINUTES);</pre>

minus	No	Restituisce una copia del parametro in input sottratto del tempo specificato.	Instant dieciMinutiFa = Instant.now().minus(10, ChronoUnit.MINUTES);
get	No	Ritorna una parte dei valori del primo parametro specificato.	DayOfWeek dayOfWeek = date.getDayOfWeek();
is	No	Ritorna lo stato dell'oggetto specificato.	boolean isLeapYear = Year.of(2016).isLeap();
to	No	Converte l'oggetto corrente in un altro tipo.	
at	No	Restituisce un nuovo oggetto ottenuto aggiungendo le informazioni specificate all'oggetto corrente.	ZonedDateTime zdt = ldt.atZone(losAngeles);
parse	Sì	Analizza la stringa in input per generare un'istanza della classe su cui è chiamato il metodo.	LocalDate localDate = LocalDate.parse(data, DateTimeFormatter.ISO_LOCAL_DATE);
format	No	Ritorna una stringa formattata usando il formatter specificato a partire dai valori nell'oggetto Temporal su cui è chiamato il metodo.	LocalDate.now().format(DateTimeFormatter.ISO_LOCAL_DATE)

13.2.2 Il package `java.time`

Questo package contiene definizioni di tempi, durate, date e istanti. A differenza delle classi storiche `Date` e `Calendar` queste entità non sono basate sul tempo di sistema, bensì dal valore assoluto chiamato **timeline**, ovvero la **linea di tempo** misurata in nanosecondi che partono dal primo istante del primo gennaio del 1970.

13.2.2.1 La classe `Instant`

Una delle classi più importanti del package `java.time` è la classe `Instant`. Essa rappresenta un

istante nel tempo e internamente è rappresentato con i nanosecondi passati dalla mezzanotte del primo gennaio del 1970. Questa data simbolica è rappresentata con la costante statica di tipo `Instant` chiamata `EPOCH` definita all'interno della classe `Instant` stessa. Gli oggetti `Instant` che rappresentano istanti precedenti ad `EPOCH` saranno rappresentati da un numero negativo, mentre quelli successivi ad `EPOCH` saranno rappresentati da un numero positivo.

Le altre due costanti statiche definite dalla classe `Instant` sono `MIN` e `MAX` che astraggono rispettivamente il minimo `Instant` e il massimo `Instant` rappresentabili con questa classe. In particolare `MIN` rappresenta la data del primo Gennaio di -1000000000 alle 00:00, mentre `MAX` rappresenta l'istante 31 Dicembre di 1000000000 alle ore 23:59:599999999, e questo intervallo di tempo dovrebbe risultare sufficiente per ogni scopo.

`Instant` definisce metodi per sommare e sottrarre tempo di tipo “plus” e “minus”. Per esempio:

```
Instant traDieciMinuti = Instant.now().plus(10, ChronoUnit.MINUTES);
```

aggiunge dieci minuti all'istante attuale (`ChronoUnit` definisce le unità di misura temporali). Infatti il metodo `now()` restituisce l'istante attuale, e su quest'oggetto abbiamo invocato il metodo `plus()` che ha preso come primo parametro in input il valore intero 10 e come secondo parametro l'elemento `MINUTES` dell'enumerazione `ChronoUnit` del package `java.time.temporal`. Quest'ultimo è servito per specificare l'unità di misura temporale del primo parametro. Se avessimo usato `ChronoUnit.HOURS` avremmo aggiunto 10 ore.

La classe `Instant` definisce anche altri interessanti metodi come `isAfter()`, questo restituisce un booleano che indica se l'oggetto `Instant` su cui stiamo chiamando il metodo è successivo all'oggetto `Instant` specificato in input. Per esempio:

```
Instant now = Instant.now();
boolean b = now.isAfter(Instant.EPOCH);
```

la variabile `b` varrà `true`. Equivalentemente esiste il metodo `isBefore()`. Altro metodo degno di nota è il metodo `until()`. Per esempio per sapere quanti giorni avete vissuto sino a questo momento, dobbiamo istanziare un oggetto di tipo `Instant` che rappresenta l'istante della vostra nascita. Per esempio sfruttando il metodo `parse()` nel seguente modo:

```
Instant nascita = Instant.parse("2004-04-14T07:00:00.00Z");
```

(più avanti parleremo di formattazioni e parsing di date). Una volta ottenuto l'oggetto che abbiamo chiamato `nascita`, con la seguente sintassi si ottiene il numero di giorni passati dall'evento nascita ad oggi:

```
long giornoDallaNascita = nascita.until(Instant.now(), ChronoUnit.DAYS);
```

13.2.2.2 Le classi `Duration` e `Period`

La classe `Duration` rappresenta un intervallo di tempo. A differenza della classe `Instant`,

`Duration` non è connessa alla linea temporale definita a partire da `Instant.EPOCH`, bensì è basata sul tempo gestito dalla macchina dove esegue il programma. Questa classe memorizza al suo interno l'intervallo di tempo calcolato in secondi e nanosecondi. Inoltre, l'unità di tempo più grande a cui è possibile convertire l'intervallo calcolato, è il giorno della durata precisa di 24 ore (senza tener conto di ora solare, fuso orario o altri fattori esterni). Possiede il metodo statico `between()` che restituisce l'oggetto `Duration` compreso tra i due `Instant` specificati come parametro del metodo:

```
Duration duration = Duration.between(nascita, Instant.now());
```

Inoltre definisce diversi metodi come `toDays()`, `toHours()`, `toSeconds()`, `toMillis()`, `toNanos()`, ed altre decine di metodi di utilità per avere informazioni su periodi di tempo.

La classe `Duration` si differenzia dalla classe `Period`, in quanto quest'ultima si basa su unità temporali quali anni, mesi e giorni. Definisce i metodi `getYears()`, `getMonths()` e `getDays()` per recuperare le informazioni. Per esempio, se vogliamo contare quanto tempo è passato dal nostro compleanno possiamo scrivere:

```
Period period = Period.between(mioCompleanno, oggi);
System.out.printf("Hai %s anni, %s mesi e %s giorni ",
    period.getYears(), period.getMonths(), period.getDays());
```

Un altro modo per calcolare un intervallo di tempo è offerto dal metodo `between()` definito nell'enumerazione `ChronoUnit` (cfr. paragrafo 13.2.4.1).

13.2.2.3 La enumerazioni `DayOfWeek` e `Month`

Esistono delle semplici enumerazioni nel package `java.time`: `DayOfWeek` e `Month` i cui elementi rappresentano semplicemente i giorni della settimana e i mesi dell'anno. I sette elementi di `DayOfWeek` si chiamano `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` e `SUNDAY`, mentre i dodici elementi di `Month` sono `JANUARY`, `FEBRUARY`, `MARCH`, `APRIL`, `MAY`, `JUNE`, `JULY`, `AUGUST`, `SEPTEMBER`, `OCTOBER`, `NOVEMBER` e `DECEMBER`. Associati a questi valori internamente ci sono dei valori interi che partono da 1 e che si possono recuperare tramite il metodo `getValue()` definito per entrambe le enumerazioni. Per lo standard ISO-8601 la numerazione è la stessa che ci aspettiamo noi italiani con `DayOfWeek.MONDAY` e `Month.JANUARY` che hanno valore 1. La numerazione degli elementi delle due enumerazioni prosegue con `DayOfWeek.TUESDAY` e `Month.FEBRUARY` con valore 2 e così via, sino a `DayOfWeek.SUNDAY` con valore 7 e `Month.DECEMBER` con valore 12.

Ci sono altri posti nel mondo dove il primo giorno della settimana è considerato domenica. Questo tipo di numerazione non è supportato dall'enumerazione `DayOfWeek`. Se comunque ci fosse un'esigenza simile, è possibile utilizzare la classe `WeekFields` che supporta la numerazione localizzata dei giorni della settimana, delle settimane del mese e delle settimane dell'anno.

13.2.2.4 Date classes: LocalDate, YearMonth, MonthYear e Year

Il package `java.time` comprende anche le “Date Classes”, ovvero classi che rappresentano date senza orario e fusi orari: `LocalDate`, `YearMonth`, `MonthDay`, e `Year`.

La classe `LocalDate` rappresenta una data senza l’informazione del tempo. Per esempio con il metodo `of()` possiamo ottenerne un’istanza in questo modo:

```
LocalDate date = LocalDate.of(2004, Month.NOVEMBER, 12);
```

con la quale è poi possibile, per esempio, recuperare il giorno della settimana nel seguente modo:

```
DayOfWeek dayOfWeek = date.getDayOfWeek();
```

Per quanto riguarda `YearMonth` rappresenta il mese di un anno. Per esempio il seguente frammento di codice:

```
YearMonth yearMonth = YearMonth.now();
YearMonth leapFebruary = YearMonth.of(2012, Month.FEBRUARY);
System.out.println("Quest'anno dura: " + yearMonth.lengthOfYear() + " giorni");
System.out.println("Il mese di Febbraio 2012 è durato " + leapFebruary.lengthOfMonth() + " giorni");
```

stamperà:

```
Quest'anno dura: 365 giorni
Il mese di Febbraio 2012 è durato 29 giorni
```

Infatti con la prima riga abbiamo istanziato un oggetto mediante il solito metodo `now()`. Con la seconda riga invece abbiamo istanziato un oggetto `YearMonth` che rappresenta il mese dell’anno 2012 che era bisestile (“leap” in inglese). Poi abbiamo stampato il numero di giorni dell’anno della variabile `yearMonth` sfruttando il metodo `lengthOfYear()` e la lunghezza del mese di Febbraio 2012.

La classe `MonthDay` invece rappresenta il giorno del mese, mentre `Year` rappresenta un anno. Sono rappresentazioni parziali delle date. Per esempio con `MonthDay` è possibile rappresentare una certa data senza specificarne l’anno:

```
MonthDay md = MonthDay.of(Month.JULY, 29);
LocalDate ld = md.atYear(2012);
```

Con il metodo statico `of()` se ne ottiene un’istanza. Con il metodo `atYear()` si aggiunge l’informazione dell’anno per ottenere un’istanza di `LocalDate`.

Con `Year` è possibile rappresentare solo l’anno senza altri dettagli:

```
Year year = Year.of(2014);
```

Queste classi hanno tantissimi metodi di utilità classici come i metodi `of()`, `with()`, `now()`, `isBefore()`, `isAfter()` etc.

13.2.2.5 Le classi `LocalTime` e `LocalDateTime`

La classe `LocalTime` invece è l'equivalente classe di `LocalDate`, ma riporta solo informazioni sul tempo e non sulla data. Eseguendo in questo momento questa istruzione:

```
System.out.println("Ora: " + LocalTime.now());
```

viene stampata l'ora esatta:

```
Ora: 23:04:13.263
```

La classe `LocalDateTime` altro non è che la classe che rappresenta sia le informazioni della data sia del tempo, ed è una delle classi più importanti della libreria. È possibile pensare a lei come una combinazione tra `LocalDate` e `LocalTime`, tanto che tra i tanti metodi di tipo “of” ce n’è uno che prende proprio un `LocalDate` e un `LocalTime`:

```
LocalDateTime localDateTime = LocalDateTime.of(LocalDate.now(),  
                                              LocalTime.now());
```

Anche in questa classe troviamo tanti metodi per creare nuove istanze aggiungendo (metodi “plus”) o sottraendo (metodi “minus”) tempi e date.

13.2.2.6 Geolocalizzazione: le classi `zonedDateTime`, `ZoneId` e `ZoneOffset`

Nel package `java.time` esistono poi classi legate ai concetti di localizzazione. Le prime due che vediamo sono `ZoneId` e `ZoneOffset`. In particolare `ZoneId` astrae il concetto di `TimeZone`, ovvero un’area geografica che condivide lo stesso orario. Solitamente è univocamente individuata da una coppia del tipo “regione/città”, per esempio “Europa/Berlin”, “America/Detroit” oppure “Africa/Kinshasa”. Per ottenere lo `ZoneId` di default per il nostro sistema è sufficiente chiamare il metodo `systemDefault()`:

```
ZoneId zoneId = ZoneId.systemDefault();
```

Per ottenere gli `ZoneId` disponibili esiste invece il metodo `getAvailableZoneIds()`:

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();
```

La classe `ZoneOffset` invece, astrae il concetto di fuso orario come differenza oraria rispetto all’orario ufficiale di Greenwich/UTC (UTC è l’orario ufficiale di Greenwich senza ora legale).

Con il seguente codice capiamo qual è il fuso orario di Los Angeles:

```
LocalDateTime ldt = LocalDateTime.now();
ZoneId losAngeles = ZoneId.of("America/Los_Angeles");
ZonedDateTime zdt = ldt.atZone(losAngeles);
ZoneOffset offset = zdt.getOffset();
```

Abbiamo prima istanziato il `LocalDateTime` attuale. Successivamente con il metodo `of` abbiamo recuperato lo `ZoneId` di Los Angeles, e tramite il metodo `atZone()` abbiamo ottenuto un'istanza di `ZonedDateTime`, che è la versione localizzata di `LocalDateTime`. Su di essa abbiamo chiamato il metodo `getOffset()` che ci ha restituito l'oggetto `ZoneOffset`.

La classe `ZonedDateTime` ha in comune con `LocalDateTime` la maggior parte dei metodi. Esistono anche la classe `OffsetDateTime`, che è equivalente a `ZonedDateTime` ma non contiene informazioni sullo `ZoneId`, e la classe `OffsetTime` che non contiene neanche le informazioni sulla data ma solo sull'offset del tempo.

Molte classi quindi definiscono il metodo `now()`, che restituisce un data basata sull'orologio del sistema su cui viene eseguita l'istruzione e il default time zone. Se per esempio dobbiamo costruire un'applicazione internazionalizzata dove l'ora può cambiare a seconda della nazione . . . dove ci si trova, allora potrebbe essere necessario utilizzare l'overload del metodo `now()` che prende in input un oggetto di tipo `Clock`. Si noti che questa classe è astratta e per istanziarla bisogna utilizzare i propri metodi statici (che ritroneranno oggetti istanziati da sottoclassi concrete di `Clock`). Una buona tecnica per utilizzare l'oggetto `Clock` è descritta nella pagina dedicata della documentazione ufficiale.

13.2.3 Il package `java.time.format`

Il package `java.time.format` è essenzialmente definito da classi che consentono il parsing, ovvero la traduzione da stringa a un oggetto concreto ma solo dopo averla analizzata, o il processo inverso, ovvero la formattazione a stringa leggibile da un oggetto che rappresenta una data/orario. Visto che una data può essere scritta in tanti modi diversi, è naturale ci sia bisogno di un package dedicato a tali funzionalità.

Praticamente la classe regina di questo package è la classe `DateTimeFormatter`, che deve essere considerata come la classe che sostituisce la vecchia classe `java.text.DateFormat`.

Questa tra le altre cose, definisce diversi metodi predefiniti per formattare le date e gli orari in formati standard. Per esempio con la seguente sintassi possiamo eseguire il parsing di una data sotto forma di stringa con formato “aaaa-mm-gg” (in inglese “yyyy-MM-dd”):

```
String data = "1942-07-22";
LocalDate           localDate          =
                    DateTimeFormatter.ISO_LOCAL_DATE;
                                         =                               LocalDate.parse(data,
```

Quindi l'oggetto predefinito `DateTimeFormatter.ISO_LOCAL_DATE` permette di interpretare stringhe con il formato “`yyyy-MM-dd`”. Esistono tanti altri formattatori predefiniti che possono essere consultati direttamente dalla documentazione della classe `DateTimeFormatter` o all'indirizzo `internet : http://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#predefined`.

È possibile anche sfruttare quest'oggetto per formattare con un certo formato una stringa, come si può osservare nel seguente frammento di codice:

```
System.out.println(DateTimeFormatter.ISO_LOCAL_DATE.format(LocalDate.now()));
```

dove andiamo a stampare con il formato specificato la data odierna a partire da un oggetto `LocalDate`. Inoltre anche tutte le classi studiate di `java.time` (come appunto `LocalDate`) definiscono un metodo `format()` che prende in input un `DateTimeFormatter`. Quindi la seguente riga produce lo stesso output dell'esempio precedente:

```
System.out.println(LocalDate.now().format(DateTimeFormatter.ISO_LOCAL_DATE));
```

Oltre ad utilizzare formattatori predefiniti, è possibile anche usare dei formattatori che dipendono dalla localizzazione. Per esempio con il seguente frammento di codice che usa anche un metodo “`with`”:

```
DateTimeFormatter dateDateTimeFormatter =  
    DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG);  
dateDateTimeFormatter = dateDateTimeFormatter.withLocale(Locale.FRENCH);  
System.out.println(dateDateTimeFormatter.format(LocalDate.now()));
```

abbiamo ottenuto un `DateTimeFormatter` mediante la chiamata al metodo `ofLocalizedDate()` a cui abbiamo passato l'elemento `LONG` dell'enumerazione `FormatStyle`. In questo modo avremo una formattazione “estesa” della data localizzata. Non avendo specificato un `Locale`, è assunto che sarà preso il `Locale` della macchina attuale (presumibilmente italiano). Con la seconda riga in cui usiamo il metodo `withLocale()`, modifichiamo il `Locale` di default con il `Locale` francese, e infine stampiamo la data attuale formattata secondo il `Locale` francese.

Infine è possibile anche creare pattern personalizzati per interpretare stringhe già in un certo formato, o formattare come stringhe delle date/orari a seconda della necessità. Per esempio, con il seguente codice:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MMM/dd/yy hh:mm  
a");  
String formattedDate = formatter.format(LocalDateTime.now());  
System.out.printf(formattedDate);
```

abbiamo deciso noi il pattern con il quale stampare la data. Si noti che per esempio specificando `MMM` formattiamo il mese con le prime tre lettere del nome, e non come numero. Mentre con “`a`” abbiamo aggiunto l'informazione AM/PM. Tutti i simboli e le lettere che vengono interpretati dai formattatori sono descritti sempre nelle pagine di documentazione della classe `DateTimeFormatter` o

all’indirizzo internet:

<http://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns>

È molto importante gestire eventuali eccezioni che possono essere lanciate quando “parsiamo” o formattiamo date e orari. Ma in fondo le regole sono banali: se formattiamo dobbiamo gestire la `DateTimeException`, mentre se “parsiamo” una stringa dobbiamo gestire una `DateTimeParseException`.

13.2.4 Il package `java.time.temporal`

Questo package contiene classi, interfacce ed enumerazioni essenzialmente dedicate a gestire calcoli su date e orari.

13.2.4.1 L’enumerazione `ChronoUnit`

Abbiamo già incontrato l’enumerazione `ChronoUnit` nella tabella del paragrafo 13.2.1. Questa definisce con i suoi elementi le varie unità di misure temporali utilizzate dalle classi della libreria “Date-Time API”, come `NANOS`, `MILLIS`, `MINUTES`, `DAYS`, `WEEKS`, `CENTURIES`, `MILLENNIA`, `DECades`, `SECONDS` e così via.

Questa enumerazione definisce anche un interessante metodo `between()` che restituisce il numero di giorni, oppure ore, oppure anni e così via (a seconda dell’elemento dell’enumerazione usato) tra due momenti temporali. Per esempio il seguente frammento di codice:

```
ChronoUnit.DAYS.between(data1, data2);
```

restituisce il numero di giorni totali passati tra `data1` e `data2`.

13.2.4.2 Temporal Adjusters

In particolare le cosiddette “Date Adjusters” forniscono metodi statici che gestiscono tipiche operazioni di calcolo di date e orari, i cui risultati possono essere passati a metodi `with()` per oggetti che rappresentano date e orari come `LocalDate` e `LocalTime`. Per esempio nel seguente frammento di codice, sfruttiamo la classe `TemporalAdjusters` (notare la “`s`” finale) per conoscere la prossima domenica:

```
LocalDate date = LocalDate.now();
System.out.printf("Prossima %s domenica: %s", date.with(TemporalAdjusters.nextOrSame(DayOfWeek.SUNDAY)));
```

Con il metodo `nextOrSame()` abbiamo trovato la domenica successiva ad oggi (o oggi stesso se oggi fosse domenica). La classe `TemporalAdjusters` contiene altre decine di metodi di utilità per

esempio per cercare l'ultimo giorno del mese, l'ultima occorrenza di un certo giorno settimanale in un mese, il precedente mercoledì e così via.

È anche possibile creare oggetti “adjuster” personalizzati creando classi che implementano l'interfaccia `TemporalAdjuster` (si noti che non stiamo parlando della classe `TemporalAdjusters`) e ridefiniscono il metodo `adjustInto()`. Per esempio supponiamo di avere l'esigenza di voler pubblicare un nuovo post sul nostro blog ogni 4 giorni. Supponiamo anche di voler evitare di pubblicare di domenica per qualche ragione. Allora potremmo creare un nostro “adjuster” con la seguente implementazione:

```
public class NextBlogPostAdjuster implements TemporalAdjuster {  
    public Temporal adjustInto(Temporal input) {  
        LocalDate data = LocalDate.from(input);  
        LocalDate result = data.plusDays(4);  
        DayOfWeek giornoDellaSettimana = result.getDayOfWeek();  
        if (giornoDellaSettimana.equals(DayOfWeek.SUNDAY)) {  
            result = result.plusDays(1);  
        }  
        return result;  
    }  
}
```

L'algoritmo è molto semplice: si prova ad aggiungere quattro giorni alla data in input, e nel caso il risultato della computazione (la variabile `result`) coincide con una domenica allora aggiungiamo un ulteriore giorno.

Si noti che l'interfaccia `Temporal` è implementata da tutte le classi che hanno a che fare con il tempo come `Instant`, `LocalDateTime`, `LocalDate`, `LocalTime` e così via.

13.2.4.3 *Temporal Queries*

La classe `TemporalQueries` definisce un insieme di query (oggetti `TemporalQuery`) predefinite per ottenere informazioni quando l'applicazione non può risalire al tipo temporale che sta usando. Per esempio possiamo usare il metodo `precision()` per ottenere la precisione per il tipo di dato che stiamo utilizzando, ovvero restituisce la più piccola `ChronoUnit` del tipo di dato che viene interrogato. Per esempio il seguente metodo:

```
public static void printPrecision(Temporal temporal) {  
    TemporalQuery<?> query = TemporalQueries.precision();  
    System.out.println(temporal.query(query));  
}
```

ricavato l'oggetto `TemporalQuery` mediante il metodo `precision()` chiamato su `TemporalQueries`, va a stamparne la precisione. Se fosse chiamato con le seguenti istruzioni:

```
printPrecision(LocalDate.now());  
printPrecision(LocalTime.now());
```

stamperebbe:

```
Days  
Nanos
```

Come con gli “adjusters” anche in questo caso è possibile creare personalizzazioni di query, implementando l’interfaccia `TemporalQuery` e ridefinendo il metodo `queryFrom()`. Per esempio la seguente classe implementa una query per capire se il giorno è un giorno del weekend:

```
public class IsWeekendQuery implements TemporalQuery<Boolean> {  
    public Boolean queryFrom(TemporalAccessor input) {  
        LocalDate data = LocalDate.from(input);  
        DayOfWeek giornoDellaSettimana = data.getDayOfWeek();  
        return (giornoDellaSettimana.equals(DayOfWeek.SATURDAY) ||  
                giornoDellaSettimana.equals(DayOfWeek.SUNDAY));  
    }  
}
```

Per prima cosa notiamo che abbiamo parametrizzato con `Boolean` l’interfaccia implementata. In questo modo il metodo `queryFrom()` ritornerà un `Boolean`. Inoltre si noti che abbiamo ritornato `true` o `false`, ma che per l’autoboxing sono compatibili con `Boolean`. Per usare la nostra query personalizzata possiamo scrivere:

```
LocalDate now = LocalDate.now();  
boolean isWeekend = now.query(new IsWeekendQuery());  
System.out.println(isWeekend);
```

che stamperà il risultato.

Le interfacce `TemporalQuery` e `TemporalAdjuster` sono interfacce funzionali, ovvero che posseggono un solo metodo astratto. Questo significherà che potremo utilizzare queste interfacce quando studieremo le espressioni lambda nel modulo 15.

Si noti che i metodi di `TemporalAdjuster` e quelli di `TemporalQueries` sono statici, e si adattano bene ad essere utilizzati con gli import statici.

13.2.5 Codice legacy

Con Java 8 e la nuova libreria “Date-Time API”, sono arrivati dei metodi per interagire con il codice scritto in precedenza (codice legacy). In particolare con le vecchie classi temporali di `java.util` come `Date`, `Calendar`, `GregorianCalendar` e `TimeZone`, c’è la possibilità di migrare alle nuove con uno sforzo tutto sommato modesto. Sono state apportate diverse modifiche alle classi di `java.util`, ne vediamo alcune di seguito.

1. È stato introdotto il metodo `toInstant()` sia nella classe `Calendar` sia nella classe `Date`.

Questo metodo ritorna un oggetto `Instant` a partire da un oggetto `Calendar` o da un oggetto `Date`. Per esempio sono valide le seguenti linee di codice:

```
Instant instant = new Date().toInstant();
Calendar calendar = Calendar.getInstance();
instant = calendar.toInstant();
```

2. Alla classe `Date` è stato aggiunto il metodo `from()` che crea un'oggetto `Date` a partire da un `Instant`. Per esempio è lecito scrivere:

```
Date date = Date.from(instant);
```

3. Nella classe `GregorianCalendar` sono stati aggiunti i metodi `toZonedDateTime()` che restituisce un oggetto `ZonedDateTime` a partire da un oggetto di tipo `GregorianCalendar`, ed il metodo `from()` che prende in input un oggetto `ZonedDateTime` per creare un `GregorianCalendar`. Per esempio è possibile scrivere:

```
GregorianCalendar gc = GregorianCalendar.from(ZonedDateTime.now());
ZonedDateTime zdt = gc.toZonedDateTime();
```

4. Infine è stato aggiunto alla classe `TimeZone` il metodo `toZoneId()`, che restituisce un oggetto di tipo `ZoneId` a partire da un oggetto `TimeZone`. Segue un esempio:

```
TimeZone tz = TimeZone.getDefault();
ZoneId zi = tz.toZoneId();
```

In generale è possibile sostituire la classe `Date` con la classe `Instant`, e la classe `GregorianCalendar` (a seconda dei casi) con le classi `ZonedDateTime`, `LocalTime` o `LocalDate`. Mentre `TimeZone` è sostituibile con `ZoneId` o `ZoneOffset`. Per quanto riguarda le classi di formattazione come `DateFormat`, nonostante la classe `java.time.format.DateTimeFormatter` sia più potente, è ancora possibile usarla, anche con le classi del package `java.time`, così come si faceva con le classi di `java.util`.

Riepilogo

In questo modulo abbiamo introdotto le classi principali del package `java.util`, e la nuova libreria per gestire date e orari. Abbiamo visto come l'**internazionalizzazione** in Java sia semplice, potente e facilmente implementabile tramite l'utilizzo dei `Locale` e `ResourceBundle`. Un altro argomento particolarmente semplice e utile è la gestione della configurazione mediante **file di properties**, grazie alle classi `Properties` e `Preferences`. Meno semplice è utilizzare le **espressioni regolari**, non tanto per le classi da utilizzare ma per la sintassi stessa delle espressioni, su cui esistono libri dedicati. D'altra parte le espressioni regolari rappresentano uno strumento molto potente per l'analisi dei testi, e soprattutto è oramai uno standard de facto dell'informatica. Abbiamo anche visto come la classe `StringTokenizer` consenta di analizzare il testo in modo estremamente semplice.

Nella seconda parte del modulo siamo passati a studiare la nuova libreria “**Date-Time API**”. Abbiamo visto come questa sia composta da classi che spesso sono calcolate a partire dalla “timeline”, come `Instant`. Esistono classi molto coese a rappresentare astrazioni precise come `YearOfMonth` e `DayOfWeek`. In generale le classi sono molto semplici da utilizzare, basti paragonare `LocalDate` e la vecchia classe `Date`. Una serie di metodi (o prefissi di metodi) standard in classi diverse come i metodi “`of`”, “`now`” e “`with`”. La **geolocalizzazione** poi si ottiene semplicemente usando classi geolocalizzate (`ZonedDateTime`) praticamente allo stesso modo di come si utilizzano quelle non geolocalizzate, usufruendo del supporto delle classi `ZoneId` e `ZoneOffset`.

Per **formattare ed analizzare le date** esiste la classe `DateTimeFormatter`, che da sola provvede ad ogni soluzione.

Abbiamo visto come i “**temporal adjusters**” offrano strumenti per fare calcoli con le date, mentre le “**temporal queries**” permettano di ricavare informazioni quando l’applicazione non può risalire al tipo temporale che sta usando.

Infine abbiamo dato un’occhiata a come Java 8 abbia fornito strumenti per migrare, o quantomeno fare interagire, le classi storiche per la gestione delle date ed i nuovi meccanismi definiti dalla “Date-Time API”.

Esercizi modulo 13

Esercizio 13.a) Package `java.util`, Vero o Falso:

1. La classe `Properties` estende `Hashtable` ma consente di salvare su un file le coppie chiave-valore rendendole persistenti.
2. La classe `Locale` astrae il concetto di zona.
3. La classe `ResourceBundle` rappresenta un file di properties che permette di gestire l’internazionalizzazione. Il rapporto tra nome del file e `Locale` specificato per individuare tale file consentirà di gestire la configurazione della lingua delle nostre applicazioni.
4. L’output del seguente codice:

```
 StringTokenizer st = new StringTokenizer(  
    "Il linguaggio object oriented Java", "t", false);  
 while (st.hasMoreTokens()) {  
     System.out.println(st.nextToken());  
 }
```

sarà:

```
 Il linguaggio objec  
 t  
 orien  
 t  
 ed Java
```

5. Il seguente codice non è valido:

```
Pattern p = Pattern.compile("\bb");
Matcher m = p.matcher("Magic Solution");
boolean b = m.find();
System.out.println(b);
```

6. La classe `Preferences` permette di gestire file di configurazione con un file XML.

7. La classe `Formatter` definisce il metodo `printf()`.

8. L'espressione regolare `[a]` è un quantificatore (greedy quantifier).

9. Il seguente codice:

```
Date d = new Date();
d.now();
```

crea un oggetto `Date` con la data attuale.

10. Un `SimpleDateFormat` può formattare e analizzare qualsiasi valuta grazie alla specifica di un `Locale`.

Esercizio 13.b) Date-Time API, Vero o Falso:

1. I metodi di tipo “from” consentono di recuperare un certo tipo temporale a partire da un tipo temporale con più informazioni.
2. L'unico modo per fare il parsing di una stringa per ottenere un oggetto `Instant` è passare tramite la classe `DateTimeFormatter`.
3. La classe `Duration` calcola la distanza tra due istanti, quindi è definita sulla timeline.
4. Non è possibile memorizzare informazioni sull'orario in un oggetto di tipo `LocalDate`.
5. È possibile memorizzare informazioni sulla data in un oggetto di tipo `LocalTime`.
6. È possibile memorizzare informazioni sulla data in un oggetto di tipo `ZonedDateTime`.
7. Il metodo `between()` di `ChronoUnit` restituisce un oggetto `Duration`.
8. I “temporal adjusters” possono essere passati a metodi di tipo “with” per compiere operazioni su date e orari.
9. I “temporal queries” possono essere passati a metodi di tipo “with” per recuperare informazioni su date e orari.
10. In generale è possibile sostituire la classe `Date` con la classe `Instant`.

Soluzioni esercizi modulo 13

Esercizio 13.a) Package `java.util`, Vero o Falso:

1. Vero.

- 2. Vero.**
- 3. Vero.**
- 4. Falso**, tutte le “t” non dovrebbero esserci.
- 5. Falso**, è valido ma stamperà `false`. Affinché stampi `true` l'espressione si deve modificare in “`\bb`”.
- 6. Falso**, il modo in cui vengono immagazzinati i dati persistenti è dipendente dalla piattaforma. Un oggetto `Properties` invece può usare anche file di configurazione in formato XML.
- 7. Falso**, la classe `PrintStream` definisce il metodo `printf()`, la classe `Formatter` definisce il metodo `format()`.
- 8. Falso**, è un gruppo di caratteri.
- 9. Falso**, la classe `Date` non ha un metodo `now()`. La prima riga da sola avrebbe adempiuto al compito richiesto.
- 10. Falso**, la classe `SimpleDateFormat` semplicemente non esiste. Ma esiste la classe `SimpleNumberFormat`.

Esercizio 13.b) Date-Time API, Vero o Falso:

- 1. Vero.**
- 2. Falso**, anche la stessa classe `Instant` definisce un metodo `parse()`.
- 3. Falso**, `Duration` non è connessa alla timeline visto che rappresenta un intervallo di tempo compreso tra due `Instant`.
- 4. Vero.**
- 5. Falso.**
- 6. Falso**, la classe `ZonedDateTime` semplicemente non esiste.
- 7. Falso.**
- 8. Vero.**
- 9. Falso.**
- 10. Vero.**

**Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper implementare programmi configurabili mediante file di properties e preferences (unità 13.2.1)	<input type="checkbox"/>	
Saper implementare programmi con l'internazionalizzazione (unità 13.2.2, 13.2.3)	<input type="checkbox"/>	
Saper utilizzare la classe <code>StringTokenizer</code> per suddividere stringhe (unità 13.1.4)	<input type="checkbox"/>	
Saper utilizzare le basi delle espressioni regolari (unità 13.1.6)	<input type="checkbox"/>	
Comprendere le classi <code>Date</code> e <code>Calendar</code> (unità 13.1.7)	<input type="checkbox"/>	
Comprendere e saper utilizzare la nuova libreria “Date-Time API” (unità 13.2)	<input type="checkbox"/>	

Note:

Gestione dei thread

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Saper definire multithreading e multitasking (unità 14.1).
- ✓ Comprendere la dimensione temporale introdotta dalla definizione dei thread in quanto oggetti (unità 14.2).
- ✓ Saper creare ed utilizzare thread tramite la classe `Thread` e l'interfaccia `Runnable` (unità 14.2).
- ✓ Definire che cos'è uno scheduler e i suoi comportamenti riguardo alle priorità dei thread (unità 14.3).
- ✓ Saper utilizzare i modificatori `volatile` e `synchronized` (unità 14.3, 14.4).
- ✓ Far comunicare i thread (unità 14.5).
- ✓ Conoscere il nucleo delle principali librerie sulla concorrenza (unità 14.6).

Questo modulo è dedicato alla gestione della concorrenza dei processi nella programmazione Java. Non capita spesso di dover gestire i thread “a mano” nella programmazione giornaliera perché sovente in Java (per esempio nelle tecnologie Web implementate dalla Enterprise Edition) i thread sono gestiti automaticamente. Tuttavia riteniamo indispensabile la conoscenza di tali concetti per padroneggiare la programmazione Java e comprendere molti concetti presenti nelle altre librerie. Per esempio nella documentazione troveremo spesso la dicitura `thread-safe` (come nella classe `java.util.Hashtable` che sarà trattata nel modulo relativo alle Collections). Questo modulo quindi non è dedicato alla libreria per gestire processi concorrenti. Non ci soffermeremo troppo sulle decine di classi che Java mette a disposizione per gestire i thread, bensì cercheremo di capire i concetti alla base.

Nella parte finale del modulo parleremo anche delle librerie che bisogna utilizzare per semplificare il nostro lavoro.

Avvertiamo il lettore che non ha ancora familiarità con questi argomenti che questo modulo è particolarmente complesso e richiede impegno e concentrazione particolari. Chi troverà l'argomento troppo ostico potrà eventualmente tornare a rileggerlo in un secondo momento, quando si sentirà pronto. Per mitigare la complessità, cercheremo comunque di approcciare ai concetti in maniera graduale.

14.1 Introduzione ai thread

I thread rappresentano il mezzo mediante il quale Java fa eseguire un'applicazione da più Virtual Machine contemporaneamente, allo scopo di ottimizzare i tempi del runtime. Ovviamente si tratta di un'illusione: per ogni programma in esecuzione esiste un'unica JVM e forse, un'unica CPU. Ma la CPU può eseguire codice tramite più processi all'interno della gestione della JVM per dare l'impressione di avere più processori. Con processori multi-core il multi-threading darà il meglio di sé.

L'esperienza ci ha insegnato che l'apprendimento di un concetto complesso come la gestione dei thread in Java richiede un approccio graduale, sia per quanto riguarda le definizioni sia per quanto riguarda le tecniche di utilizzo. Tale convincimento, oltre ad avere una natura empirica, nasce dall'esigenza di dover definire un thread quale oggetto della classe `Thread`, cosa che può portare il lettore facilmente a confondersi. Didatticamente, inoltre, è spesso utile adoperare schemi grafici per aiutare nella comprensione. Ciò risulta meno fattibile quando si cerca di spiegare il comportamento di più thread al runtime, poiché uno schema statico potrebbe non risultare sufficiente.

14.1.1 Definizione provvisoria di thread

Quando eseguiamo un'applicazione Java vengono eseguite le istruzioni contenute in essa in maniera sequenziale, a partire dal codice del metodo `main()`. Spesso, soprattutto in fase di debug, allo scopo di simulare l'esecuzione dell'applicazione, il programmatore immagina un cursore che scorre sequenzialmente le istruzioni, magari simulando il suo movimento con un dito che punta sul monitor. Un tool di sviluppo che dispone di un debugger grafico invece, evidenzia concretamente questo cursore. Consapevoli che il concetto risulterà familiare ad un qualsiasi programmatore, possiamo per il momento identificare un thread proprio con questo cursore immaginario. In tal modo affronteremo le difficoltà dell'apprendimento in maniera graduale.

14.1.2 Cosa significa multithreading

L'idea di base è semplice: immaginiamo di eseguire un'applicazione Java. Il nostro cursore immaginario scorrerà ed eseguirà sequenzialmente le istruzioni partendo dal codice del metodo `main()`. Quindi, a runtime, esiste almeno un thread in esecuzione, ed il suo compito è eseguire il codice, seguendo il flusso definito dall'applicazione stessa.

Per **multithreading** si intende il processo che porterà un'applicazione a definire più di un thread, assegnando ad ognuno compiti da eseguire parallelamente. Il vantaggio che può portare un'applicazione multithreaded è relativo alle prestazioni della stessa. Infatti i thread possono dialogare tra loro allo scopo di spartirsi nella maniera ottimale l'utilizzo delle risorse del sistema. D'altronde l'esecuzione parallela di più thread all'interno della stessa applicazione è vincolata all'architettura della macchina su cui esegue. In altre parole, se la macchina monta un unico processore, in un determinato momento t può essere in esecuzione un unico thread. Ciò significa che un'applicazione non multithreaded che richiede in un determinato momento alcuni input (da un utente, da una rete, da un database etc.) per poter proseguire nell'esecuzione, quando si trova in stato di attesa non può eseguire nulla. Un'applicazione multithreaded invece, potrebbe eseguire altro codice mediante un altro thread, avvertito dal thread che è in stato di attesa.

Solitamente il multithreading è una caratteristica dei sistemi operativi (per esempio Unix) piuttosto

che dei linguaggi di programmazione. La tecnologia Java, tramite la Virtual Machine, ci offre uno strato d'astrazione per gestire il multithreading direttamente dal linguaggio. Altri linguaggi (come C/C++) solitamente sfruttano le complicate librerie del sistema operativo per gestire il multithreading dove possibile. Infatti tali linguaggi, non essendo stati progettati per lo scopo, non supportano un meccanismo chiaro per gestire i thread.

Il multithreading non deve essere confuso con il multitasking. Possiamo definire task i processi “pesanti”, per esempio Word ed Excel. In un sistema operativo che supporta il multitasking è possibile eseguire più task contemporaneamente. La precedente affermazione può risultare scontata per molti lettori, ma negli anni ‘80 ...

...

l’Home Computer utilizzava spesso sistemi chiaramente non multitasking. Ed anche negli ultimi anni (vedi il primo IPAD) c’è stato qualche esempio. I task hanno spazi di indirizzi separati e la comunicazione fra loro è limitata. I thread possono essere definiti processi “leggeri”, che condividono lo stesso spazio degli indirizzi e lo stesso processo pesante (il processo padre) in cooperazione. I thread hanno quindi la caratteristica fondamentale di poter comunicare al fine di ottimizzare l’esecuzione dell’applicazione in cui sono definiti.

In Java i meccanismi della gestione dei thread risiedono essenzialmente:

- ❑ nella classe `Thread` e nell’interfaccia `Runnable` (package `java.lang`).
- ❑ Nella classe `Object` (package `java.lang`).
- ❑ Nella JVM e nella keyword `synchronized`.

14.2 La classe `Thread` e la dimensione temporale

Come abbiamo precedentemente affermato, quando si avvia un’applicazione Java c’è almeno un thread in esecuzione appositamente creato dalla JVM per eseguire il codice dell’applicazione. Nel seguente esempio introdurremo la classe `Thread` e vedremo che anche con un unico thread è possibile creare situazioni interessanti.

```
1 public class ThreadExists {  
2     public static void main(String args[]) {  
3         Thread t = Thread.currentThread();  
4         t.setName("Thread principale");  
5         t.setPriority(10);  
6         System.out.println("Thread in esecuzione: " +  
7             t.getName());  
8         try {  
9             for (int n = 5; n > 0; n--) {  
10                 System.out.println(" " + n);  
11                 t.sleep(1000);  
12             }  
13         } catch (InterruptedException e) {}  
14     }  
15 }
```

```
12      }
13      catch (InterruptedException e) {
14          System.out.println("Thread interrotto");
15      }
16  }
17 }
```

Segue l'output dell'applicazione:

```
C:\TutorialJavaThread\Code>java ThreadExists
Thread in esecuzione: Thread[Thread principale,10,main]
5
4
3
2
1
```

14.2.1 Analisi di ThreadExists

Questa semplice classe produce un risultato tutt'altro che trascurabile: la gestione del tempo. La durata dell'esecuzione del programma è infatti quantificabile in circa cinque secondi. In questo testo è il primo esempio che gestisce in qualche modo la durata dell'applicazione stessa. Analizziamo il codice nei dettagli.

Alla riga 3 viene chiamato il metodo statico `currentThread()` della classe `Thread`. Questo restituisce l'indirizzo dell'oggetto `Thread` che sta eseguendo l'istruzione e che viene assegnato al reference `t`. Questo passaggio è particolarmente delicato: abbiamo identificato un thread con il “cursore immaginario” che elabora il codice. Inoltre abbiamo anche ottenuto un reference a questo cursore. Una volta ottenuto un reference è possibile gestire il thread, controllando così l'esecuzione (temporale) dell'applicazione!

Notiamo la profonda differenza tra la classe `Thread` e tutte le altre classi della libreria standard. La classe `Thread` astrae un concetto che non solo è dinamico, ma addirittura rappresenta l'esecuzione stessa dell'applicazione. Il `currentThread` infatti non è l’“oggetto corrente”, solitamente individuato dalla parola chiave `this`, ma l'oggetto che esegue l'oggetto corrente (`thread corrente`). Potremmo affermare che un oggetto `Thread` si trova in un'altra dimensione rispetto agli altri oggetti: la dimensione temporale.

Continuiamo con l'analisi della classe `ThreadExists`. Alle righe 4 e 5 scopriamo che è possibile non solo assegnare un nome al thread, ma anche una priorità. La scala delle priorità dei thread in Java va dalla priorità minima 1 alla massima 10, e la priorità di default è 5.

Come vedremo più avanti, il concetto di priorità NON è la chiave per gestire i thread.

Infatti, limitandoci alla sola gestione delle priorità, la nostra applicazione multithreaded potrebbe comportarsi in maniera differente su sistemi diversi.

Alla riga 6 viene stampato l'oggetto `t` (ovvero `t.toString()`). Dall'output notiamo che vengono stampate informazioni sul nome e la priorità del thread, oltre che sulla sua appartenenza al gruppo dei thread denominato `main`. I thread infatti appartengono a un gruppo di thread astratto dalla classe `ThreadGroup`, ma non ci occuperemo di questo argomento in dettaglio perché relativamente poco interessante. Tra la riga 7 e la riga 12 viene dichiarato un blocco `try` contenente un ciclo `for` che esegue un conto alla rovescia da 5 ad 1. Tra una stampa di un numero ed un'altra avviene una chiamata al metodo `sleep()` sull'oggetto `t`, a cui viene passato l'intero `1000`. In questo modo il thread che esegue il codice effettuerà una pausa di circa un secondo (mille millisecondi) tra la stampa di un numero e il successivo.

Siccome l'implementazione di questo metodo si basa sul tempo fornito dal sistema operativo, non bisogna fare affidamento ciecamente sul fatto che la pausa sarà effettivamente di un secondo preciso.

Tra la riga 13 e la riga 15 viene definito il blocco `catch` il quale gestisce una `InterruptedException` che il metodo `sleep()` dichiara nella sua clausola `throws`. Questa eccezione scatterebbe nel caso in cui il thread non riuscisse ad eseguire il suo codice perché interrotto da un altro thread tramite il metodo che si chiama per l'appunto `interrupt()`. Nel nostro esempio però, non vi è che un unico thread, e quindi la gestione dell'eccezione non ha molto senso. Nel prossimo paragrafo vedremo come creare altri thread sfruttando il thread principale.

14.2.2 L'interfaccia `Runnable` e la creazione dei thread

Per avere più thread basta istanziarne altri dalla classe `Thread`. Nel prossimo esempio noteremo che, quando si istanzia un oggetto `Thread`, bisogna passare al costruttore un'istanza di una classe che implementa l'interfaccia `Runnable`. In questo modo il nuovo thread, quando sarà fatto partire (mediante la chiamata al metodo `start()`), eseguirà il codice del metodo `run()` dell'istanza associata. L'interfaccia `Runnable` quindi, richiede l'implementazione del solo metodo `run()` che definisce il comportamento di un thread, e l'avvio di un thread si ottiene con la chiamata del metodo `start()`. Dopo aver analizzato il prossimo esempio, le idee dovrebbero risultare più chiare.

```
1 public class ThreadCreation implements Runnable {
2     public ThreadCreation () {
3         Thread ct = Thread.currentThread();
4         ct.setName("Thread principale");
5         Thread t = new Thread(this, "Thread figlio");
6         System.out.println("Thread attuale: " + ct);
7         System.out.println("Thread creato: " + t);
8         t.start();
9         try {
10             Thread.sleep(3000);
```

```
11      }
12      catch (InterruptedException e) {
13          System.out.println("principale interrotto");
14      }
15      System.out.println("uscita Thread principale");
16  }
17  public void run() {
18      try {
19          for (int i = 5; i > 0; i--) {
20              System.out.println("" + i);
21              Thread.sleep(1000);
22          }
23      }
24      catch (InterruptedException e) {
25          System.out.println("Thread figlio interrotto");
26      }
27      System.out.println("uscita Thread figlio");
28  }
29  public static void main(String args[]) {
30      new ThreadCreation();
31  }
32 }
```

Segue l'output del precedente codice:

```
C:\TutorialJavaThread\Code>java ThreadCreation
Thread attuale: Thread[Thread principale,5,main]
Thread creato: Thread[Thread figlio,5,main]
5
4
3
uscita Thread principale
2
1
uscita Thread figlio
```

14.2.3 Analisi di ThreadCreation

Nel precedente esempio, oltre al thread principale, ne è stato istanziato un secondo. La durata dell'esecuzione del programma è anche in questo caso quantificabile in circa cinque secondi.

LEGENDA

Thread:Stati

creato	c	classe	
pronto	p	oggetto	
running	r		
blocked	b		
sleeping	s	dead	d

Figura 14.0 - Elementi della notazione degli schemi.

riga 5: Thread t = new Thread(this, "Thread figlio");

ThreadCreation

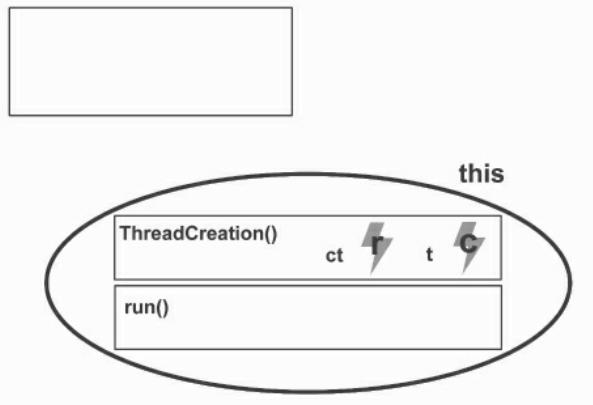


Figura 14.1 - Istanza del Thread.

Analizziamo il codice nei dettagli.

L'applicazione al runtime viene eseguita a partire dal metodo `main()` alla riga 29. Alla riga 30 viene istanziato un oggetto della classe `ThreadCreation`; poi il nostro cursore immaginario si sposta ad eseguire il costruttore dell'oggetto appena creato, alla riga 3. Qui il cursore immaginario (il thread corrente) ottiene un reference `ct`. Notiamo che "`ct` esegue `this`". A `ct` viene poi assegnato il nome "thread principale". Alla riga 5 viene finalmente istanziato un altro thread, dal thread corrente `ct` che sta eseguendo la riga 5 (Figura 14.1).

Viene utilizzato un costruttore che riceve in input due parametri. Il primo (`this`) è un oggetto `Runnable` (ovvero un'istanza di una classe che implementa l'interfaccia `Runnable`); il secondo è il

nome del thread. L'oggetto `Runnable` contiene il metodo `run()` che diventa l'obiettivo dell'esecuzione del thread `t`. Quindi, mentre per il thread principale l'obiettivo dell'esecuzione è scontato, per i thread che vengono istanziati bisogna specificarlo passando al costruttore un oggetto `Runnable`. Alle righe 6 e 7 vengono stampati messaggi descrittivi dei due thread. Alla riga 8 viene finalmente fatto partire il thread `t` mediante il metodo `start()`.

La chiamata al metodo `start()` per eseguire il metodo `run()` fa sì che il thread `t` vada prima o poi ad eseguire il metodo `run()`. Invece, una eventuale chiamata del metodo `run()` non produrrebbe altro che una normale esecuzione dello stesso da parte del thread principale `ct` e non ci sarebbe multithreading.

La partenza (tramite l'invocazione del metodo `start()`) di un thread **non** implica che il thread inizi immediatamente ad eseguire il codice, ma solo che è stato reso eleggibile per l'esecuzione (Figura 14.2).

riga 8: `t.start();`

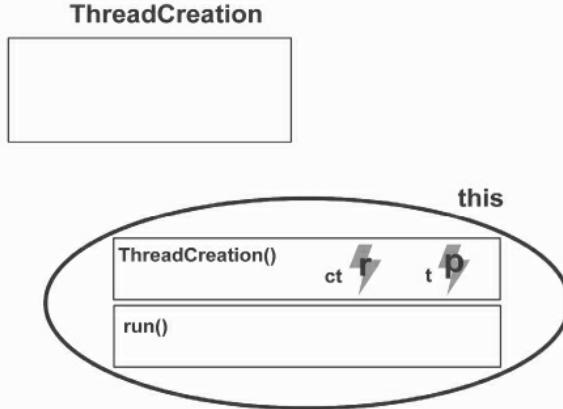


Figura 14.2 - Chiamata al metodo `start()`.

Quindi, il thread `ct`, dopo aver istanziato e reso eleggibile per l'esecuzione il thread `t`, continua nella sua esecuzione fino a quando, giunto ad eseguire la riga 10, incontra il metodo `sleep()` che lo ferma per 3 secondi.

Si noti come il metodo `sleep()` sia statico. Infatti viene reso dormiente il thread che esegue il metodo.

Thread t, riga 20 (prima iterazione): System.out.println(i);

ThreadCreation

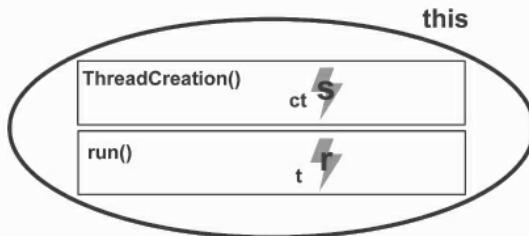


Figura 14.3 - Prima iterazione, stampa 5.

A questo punto il processore è libero dal thread `ct` e viene utilizzato dal thread `t`, che finalmente può eseguire il metodo `run()`. Ecco che allora il thread `t` esegue un ciclo, che come nell'esempio precedente realizza un conto alla rovescia, facendo pause da un secondo. Esaminando l'output verifichiamo come il codice faccia sì che il thread `t` stampi il 5 (Figura 14.3), faccia una pausa di un secondo (Figura 14.4), stampi 4, si metta in pausa per un secondo, stampi 3, effettui un'altra pausa di un secondo. Poi si risveglia il thread `ct`, che stampa la frase “uscita thread principale” (Figura 14.5) e “muore”.

Thread t, riga 21 (prima iterazione): Thread.sleep(1000);

ThreadCreation

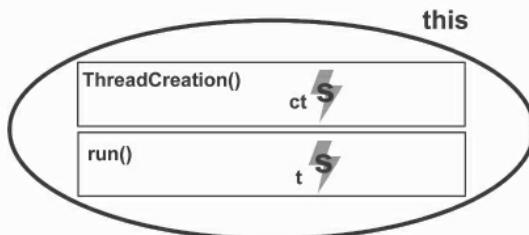


Figura 14.4 - Prima iterazione, tutti i thread in pausa.

Thread ct, riga 27: System.out.println("Uscita Thread Principale");

ThreadCreation

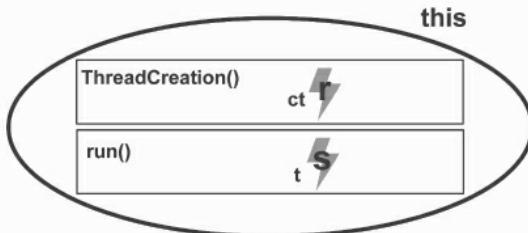


Figura 14.5 - Uscita thread principale.

Quasi contemporaneamente viene stampato 2 (Figura 14.6) cui segue una pausa di un secondo, la stampa di 1, ancora una pausa di un secondo e la stampa di “uscita thread figlio”.

Thread t, riga 20 (quarta iterazione): System.out.println(i);

ThreadCreation

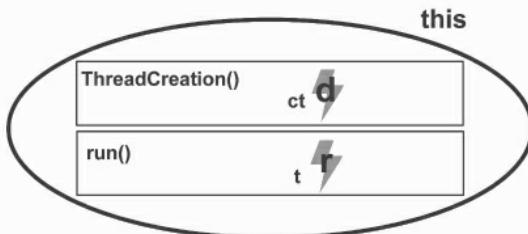


Figura 14.6 - Quarta iterazione.

Quindi nell'esempio l'applicazione ha un ciclo di vita superiore a quello del thread principale, grazie al thread creato.

Un altro metodo simile al metodo `sleep()` è il metodo `join()`. A differenza di `sleep()`

però, `join()` non è un metodo statico. Va chiamato su una particolare istanza. In particolare, supponiamo che un certo thread stia eseguendo una certa parte di codice. Se trova sulla sua strada un'espressione del tipo:

```
t.join()
```

dove `t` è un altro thread in esecuzione. Il thread corrente si metterà in pausa sino a quando il thread `t` non avrà completato il suo compito.

14.2.4 La classe `Thread` e la creazione dei thread

Abbiamo appena visto come un thread creato deve eseguire codice di un oggetto istanziato da una classe che implementa l'interfaccia `Runnable` (l'oggetto `Runnable`). Ma la classe `Thread` stessa implementa l'interfaccia `Runnable`, fornendo una implementazione vuota del metodo `run()`. È quindi possibile fare eseguire ad un thread il metodo `run()` definito all'interno dello stesso oggetto `thread`. Per esempio:

```
1 public class CounterThread extends Thread {  
2     public void run() {  
3         for (int i = 0; i<10; ++i)  
4             System.out.println(i);  
5     }  
6 }
```

Si può istanziare un thread senza specificare l'oggetto `Runnable` al costruttore, e farlo partire con il solito metodo `start()`:

```
1 CounterThread thread = new CounterThread ();  
2 thread.start();
```

È possibile (ma non consigliato) creare un thread che utilizza un `CounterThread` come oggetto `Runnable`:

```
Thread t = new Thread(new CounterThread());  
t.start();
```

Sicuramente la strategia di fare eseguire il metodo `run()` all'interno dell'oggetto `thread` stesso è più semplice rispetto a quella vista nel paragrafo precedente. Tuttavia ci sono almeno tre buone ragioni per preferire il passaggio di un oggetto `Runnable`:

1. In Java una classe non può estendere più di una classe alla volta. Quindi implementare l'interfaccia `Runnable`, piuttosto che estendere `Thread`, permetterà di utilizzare l'estensione per un'altra classe.
2. Solitamente un oggetto della classe `Thread` non dovrebbe possedere variabili d'istanza private che rappresentano i dati da gestire. Quindi il metodo `run()` nella sottoclasse di `Thread` non potrà accedere, o non potrà accedere in maniera "pulita", a tali dati.

3. Dal punto di vista della programmazione object-oriented, una sottoclasse di `Thread` che definisce il metodo `run()` combina due funzionalità poco relazionate tra loro: il supporto del multithreading ereditato dalla classe `Thread` e l'ambiente esecutivo fornito dal metodo `run()`. Quindi in questo caso l'oggetto creato è un thread associato con se stesso, e questa non è una soluzione molto object-oriented.

14.3 Priorità, scheduler e sistemi operativi

Abbiamo visto come il metodo `start()` chiamato su un thread non implichi che questo inizi immediatamente ad eseguire il suo codice (contenuto nel metodo `run()` dell'oggetto associato). In realtà la JVM definisce un thread scheduler, che si occuperà di decidere in ogni momento quale thread deve trovarsi in esecuzione. Il problema è che la JVM stessa è un software in funzione su un determinato sistema operativo e la sua implementazione dipende dal sistema. Quando si gestisce il multithreading ciò può apparire evidente, come nel prossimo esempio. Infatti lo scheduler della JVM deve comunque rispettare la filosofia dello scheduler del sistema operativo, e questa può cambiare notevolmente tra sistema e sistema. Prendiamo in considerazione solo due sistemi: Unix (Solaris) e Windows (qualsiasi versione). Il seguente esempio produce sui due sistemi output completamente diversi:

```
1 public class Clicker implements Runnable {
2     private long click = 0L;
3     private Thread t;
4     private volatile boolean running = true;
5     public Clicker(int p) {
6         t = new Thread(this);
7         t.setPriority(p);
8     }
9     public long getClick() {
10         return click;
11     }
12     public void run() {
13         while (running) {
14             click++;
15         }
16     }
17     public void stopThread() {
18         running = false;
19     }
20     public void startThread() {
21         t.start();
22     }
23 }
24 public class ThreadRace {
25     public static void main(String args[]) {
26         Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
27         Clicker hi = new Clicker(Thread.NORM_PRIORITY + 2);
28         Clicker lo = new Clicker(Thread.NORM_PRIORITY - 2);
29         lo.startThread();
30         hi.startThread();
31         try {
```

```
32         Thread.sleep(10000);
33     }
34     catch (Exception e){}
35     lo.stopThread();
36     hi.stopThread();
37     System.out.println(lo.getClick() + " vs." + hi.getClick());
38   }
39 }
```

Segue l'output su Sun Solaris 8:

```
solaris% java ThreadRace
0 vs. 1963283920
```

Vari output su Windows 8:

```
C:\TutorialJavaThread\Code>java ThreadRace
4880762242 vs. 4872062039
C:\TutorialJavaThread\Code>java ThreadRace
4850917490 vs. 4824754791
C:\TutorialJavaThread\Code>java ThreadRace
4915745356 vs. 4920787825
C:\TutorialJavaThread\Code>java ThreadRace
4942114184 vs. 4947401095
```

14.3.1 Analisi di ThreadRace

Nell'esempio si fa uso di un modificatore sino ad ora non illustrato: `volatile`, e ne parleremo nel paragrafo 14.3.4.

Il precedente esempio è composto da due classi: `ThreadRace` e `Clicker`. `ThreadRace` contiene il metodo `main()` e rappresenta quindi la classe principale. Immaginiamo di eseguire l'applicazione.

Alla riga 26 viene assegnata al thread corrente la priorità massima mediante la costante statica intera della classe `Thread` `MAX_PRIORITY`, che vale 10. Alle righe 27 e 28 vengono istanziati due oggetti dalla classe `Clicker`, `hi` e `lo`, ai cui costruttori vengono passati i valori interi 7 e 3. Gli oggetti `hi` e `lo`, tramite costruttori, creano due thread associati ai propri metodi `run()` rispettivamente con priorità 7 e priorità 3. Il thread principale continua nella sua esecuzione chiamando su entrambi gli oggetti `hi` e `lo` il metodo `startThread()`, che rende eleggibili per l'esecuzione i due thread a priorità 7 e 3 nei rispettivi oggetti (Figura 14.7).

righe 29 e 30: hi.startThread(); e lo.startThread();

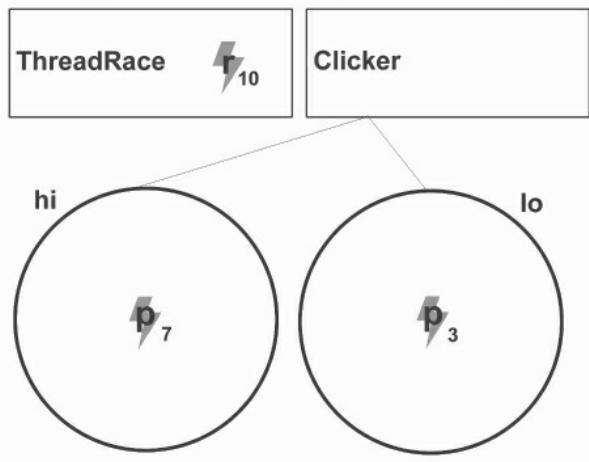


Figura 14.7 - Start dei thread.

Alla riga 32, il thread principale a priorità 10 va a dormire per una decina di secondi, lasciando disponibile la CPU agli altri due thread (Figura 14.8).

È in questo momento dell'esecuzione dell'applicazione che lo scheduler avrà un comportamento dipendente dalla piattaforma.

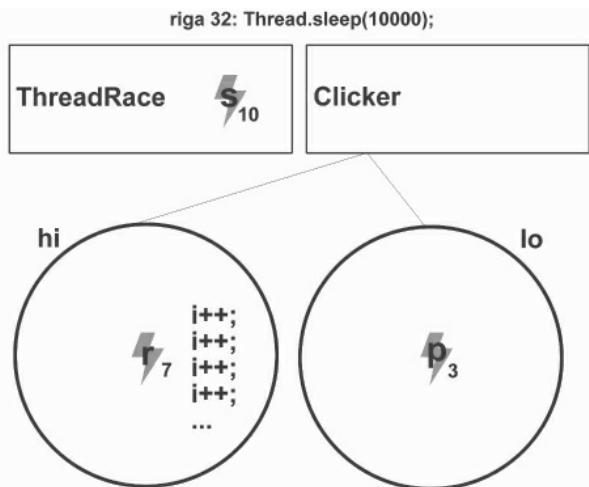


Figura 14.8 - Il thread principale si addormenta'.

14.3.2 Comportamento Windows (time slicing o round-robin

scheduling)

Un thread può trovarsi in esecuzione solo per un certo periodo di tempo, poi deve lasciare ad altri thread la possibilità di essere eseguiti. Ecco che allora l'output di Windows evidenzia che entrambi i thread hanno avuto la possibilità di eseguire codice. Il thread a priorità 7 ha avuto a disposizione per molto più tempo il processore rispetto al thread a priorità 3. Tale comportamento è però non deterministico e quindi l'output prodotto cambierà anche radicalmente ad ogni esecuzione dell'applicazione.

14.3.3 Comportamento Unix (preemptive scheduling)

Un thread in esecuzione può uscire da questo stato solo nelle seguenti situazioni:

1. viene chiamato un metodo di scheduling come `wait()` o `suspend()`.
2. Viene chiamato un metodo di blocking, come quelli dell' I/O.
3. Può essergli sospeso l'utilizzo della CPU da parte di un altro thread a priorità più alta che diviene eleggibile per l'esecuzione.
4. Termina la sua esecuzione (ovvero termina il codice del suo metodo `run()`).

Quindi il thread a priorità 7 ha occupato la CPU per tutti i 10 secondi nei quali il thread principale (a priorità 10) è in pausa. Quando l'altro thread si risveglia, rioccupa di forza la CPU.

Su entrambi i sistemi l'applicazione continua con il thread principale che, chiamando il metodo `stopThread()` (righe 35 e 36) su entrambi gli oggetti `hi` e `lo`, imposta le variabili `running` a `false` (Figure 14.9 e 14.10).

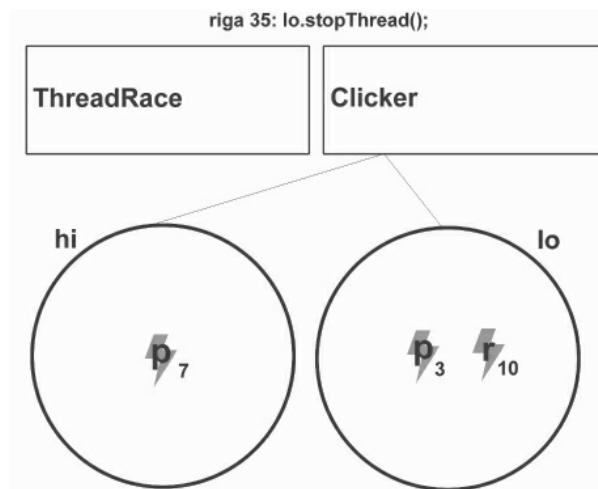


Figura 14.9 - Stop del thread `lo`.

riga 36: hi.stopThread();

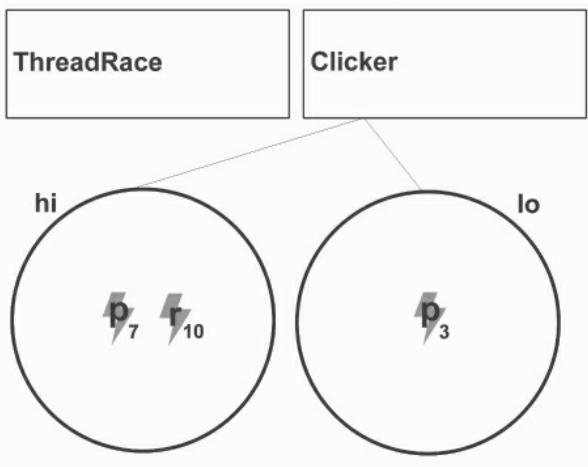


Figura 14.10 - Stop del thread hi.

Il thread principale termina la sua esecuzione stampando il risultato finale (Figure 14.11 e 14.12).

riga 37: System.out.println(io.getClick() + " vs " + hi.getClick());

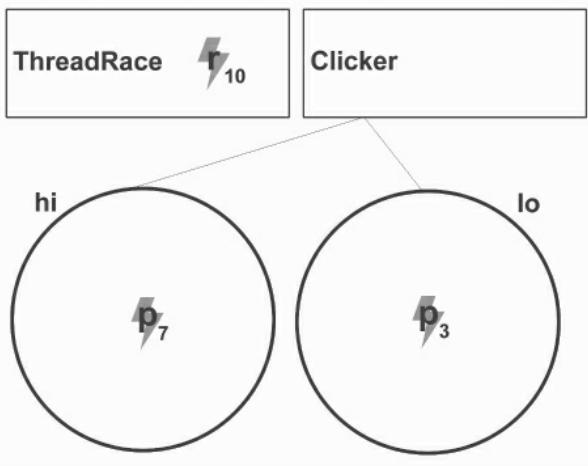


Figura 14.11 - Stampa risultato finale.

riga 38: } (fine del thread principale)

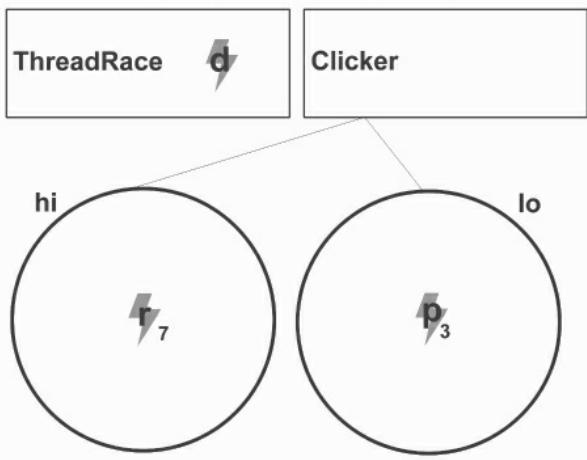


Figura 14.12 - Fine del thread principale.

A questo punto parte il thread a priorità 7, la cui esecuzione si era bloccata alla riga 13 o 14 o 15. Se riparte dalla riga 14, la variabile `click` viene incrementata un'ultima volta, senza però influenzare l'output dell'applicazione. Quando si troverà alla riga 13, la condizione del ciclo `while` non verrà verificata, quindi il thread a priorità 7 terminerà la sua esecuzione (Figura 14.13). Stessa sorte toccherà al thread a priorità 3 (Figura 14.14).

riga 18: thread a priorità 7: while (running) fallisce

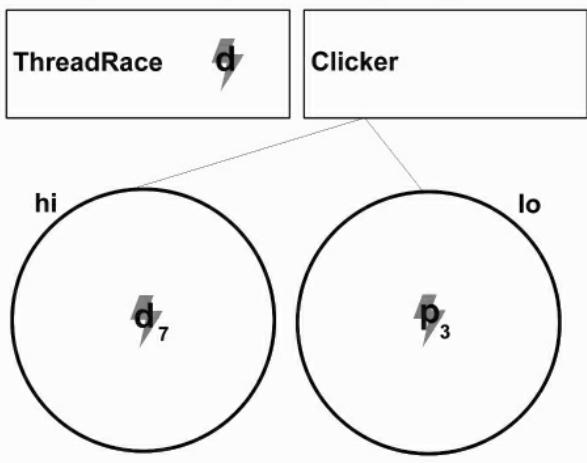


Figura 14.13 - Termina il thread hi.

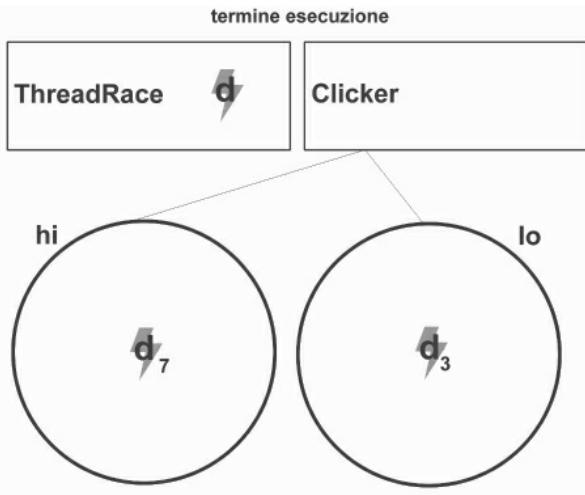


Figura 14.14 - Termina il thread lo.

14.3.4 Il modificatore `volatile`

Si tratta di un modificatore dall'utilizzo raro, che solo una percentuale ristretta di programmatore Java utilizza. Questo modificatore è applicabile solo a variabili d'istanza e per poterne apprezzare l'utilizzo dobbiamo prima fare un piccolo preambolo.

Java nel suo utilizzo della memoria esegue diverse ottimizzazioni. Tra queste c'è quella di creare copie di variabili d'istanza a cui accedono più thread, allo scopo di migliorare le prestazioni. Il modificatore `volatile` non permette questo tipo di ottimizzazione, garantendo a tutti i thread l'accesso alla stessa area di memoria. L'utilizzo di questo modificatore quindi, se da un lato può influire negativamente sulle prestazioni, in casi come quello dell'esempio precedente diventa fondamentale. Infatti dichiarando la variabile booleana `running` come `volatile`, il thread principale accede alle stesse valori della variabile `running` sia nell'oggetto `hi`, sia nell'oggetto `lo`, condividendo tali valori con i thread rispettivamente creati nel costruttore degli oggetti `hi` e `lo`. Se la variabile booleana non fosse dichiarata `volatile`, è molto probabile che il programma non funzionerebbe correttamente, perché i thread creati nei costruttori di `hi` e `lo` non leggerebbero la stessa variabile che è stata modificata dal thread principale nel metodo `stopThread()`. Il programma quindi continuerebbe a vivere con i suoi thread per continuare a "fare clic" all'infinito.

Si noti che da Java 5 in poi, l'accesso ad una variabile `volatile` garantisce che siano aggiornate anche tutte le altre variabili d'istanza `non volatile`, modificate dallo stesso thread. In questo esempio quindi, anche dichiarando la variabile `click` come `volatile` in luogo di `running`, avremmo ottenuto lo stesso risultato.

Non è necessario dichiarare `volatile` una variabile che non sarà acceduta da più thread (ma a volte

non lo si sa a priori).

Un altro effetto dell'uso del modificatore `volatile`, è che garantisce che gli accessi in scrittura o lettura sulla variabile siano atomici, ovvero che mentre un thread legge o scrive il valore di una variabile, un altro thread deve aspettare che l'operazione si concluda per poter accedere al valore della variabile. In realtà questo è già vero per tutti i tipi primitivi a parte `long` e `double`, ma l'uso del modificatore estende anche a questi due tipi questa peculiarità. Attenzione che operazioni con gli operatori di pre e post-incremento (o decremento) non sono operazioni atomiche, in quanto composte da due sotto-operazioni (aggiornamento e assegnazione) cfr. modulo 4. L'atomicità in questi casi va garantita con un altro modificatore, il modificatore `synchronized` che spiegheremo nel prossimo modulo, oppure usando la libreria `java.util.concurrent.atomic`, come vedremo nel paragrafo 14.6.3.2.

14.4 Thread e sincronizzazione

Abbiamo sino ad ora identificato un thread come un cursore immaginario. Per quanto sia utile, riteniamo il momento maturo per poter dare una definizione più scientifica di thread.

Definizione: un thread è un **processore virtuale** che esegue **codice** su determinati **dati**.

Nell'esempio precedente, `ThreadRace`, i due thread creati (quelli a priorità 7 e 3) eseguono lo stesso codice (il metodo `run()` della classe `Clicker`) utilizzando dati diversi (le variabili `lo.click` e `hi.click`). Quando però due o più thread necessitano contemporaneamente dell'accesso ad una fonte di dati condivisa, bisogna che accedano ai dati uno alla volta; cioè i loro metodi vanno sincronizzati (`synchronized`). Consideriamo l'esempio seguente:

```
1 class CallMe {
2     /*synchronized*/ public void call(String msg) {
3         System.out.print("[ " + msg);
4         try {
5             Thread.sleep(1000);
6         }
7         catch (Exception e){}
8         System.out.println(" ]");
9     }
10 }
11 class Caller implements Runnable {
12     private String msg;
13     private CallMe target;
14     public Caller(CallMe t, String s) {
15         target = t;
16         msg = s;
17         new Thread(this).start();
18     }
19     public void run() {
20         //synchronized(target) {
21             target.call(msg);
22         //}
23     }
24 }
25 public class Synch {
26     public static void main(String args[]) {
```

```
27     CallMe target = new CallMe();
28     new Caller(target, "Hello");
29     new Caller(target, "Synchronized");
30     new Caller(target, "World");
31 }
32 }
```

Di cui l'output senza sincronizzazione è:

```
[Hello[Synchronized[World]
]
]
```

e l'output con sincronizzazione:

```
[Hello]
[Synchronized]
[World]
```

14.4.1 Analisi di Synch

Nell'esempio ci sono tre classi: `Synch`, `Caller`, `CallMe`. Immaginiamo il runtime dell'applicazione e partiamo dalla classe `Synch` che contiene il metodo `main()`.

Il thread principale alla riga 27 istanzia un oggetto chiamato `target` dalla classe `CallMe`. Alla riga 28 istanzia (senza referenziarlo) un oggetto della classe `Caller`, al cui costruttore passa l'istanza `target` e la stringa `Hello`. A questo punto il thread principale si è spostato nell'istanza appena creata della classe `Caller`, per eseguirne il costruttore (righe da 14 a 18). In particolare impone come variabili d'istanza sia l'oggetto `target` sia la stringa `Hello`, quest'ultima referenziata come `msg`. Inoltre crea e fa partire un thread al cui costruttore viene passato l'oggetto `this`. Poi il thread principale ritorna alla riga 29 istanziando un altro oggetto della classe `Caller`. Al costruttore di questo secondo oggetto vengono passate la stessa istanza `target` che era stata passata al primo, e la stringa `msg`. Prevedibilmente il costruttore di questo oggetto appena istanziato imposta le variabili d'istanza `target` e `msg` (con la stringa `Synchronized`), e creerà e farà partire un thread il cui campo d'azione sarà il metodo `run()` di questo secondo oggetto `Caller`. Infine, il thread principale creerà un terzo oggetto `Caller`, che avrà come variabili d'istanza sempre lo stesso oggetto `target` e la stringa `World`. Anche in questo oggetto viene creato e fatto partire un thread, il cui campo d'azione sarà il metodo `run()` di questo terzo oggetto `Caller`. Subito dopo il thread principale muore e lo scheduler dovrà scegliere quale dei tre thread in stato ready (pronto per l'esecuzione) dovrà essere eseguito (Figura 14.15).

Avendo tutti i thread la stessa priorità 5 di default, verrà scelto quello che è da più tempo in attesa. Ecco che allora il primo thread creato (nell'oggetto dove `msg = "Hello"`) eseguirà il proprio metodo `run()` chiamando il metodo `call()` sull'istanza `target`. Quindi il primo thread si sposta nel metodo `call()` dell'oggetto `target` (righe da 2 a 9), mettendosi a dormire per un secondo (Figura 14.17) dopo aver stampato una parentesi quadra d'apertura e la stringa `Hello` (Figura 14.16).

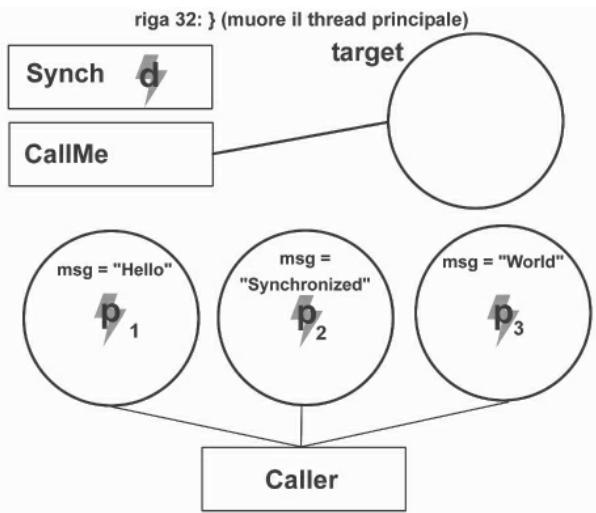


Figura 14.15 - Muore il thread principale.

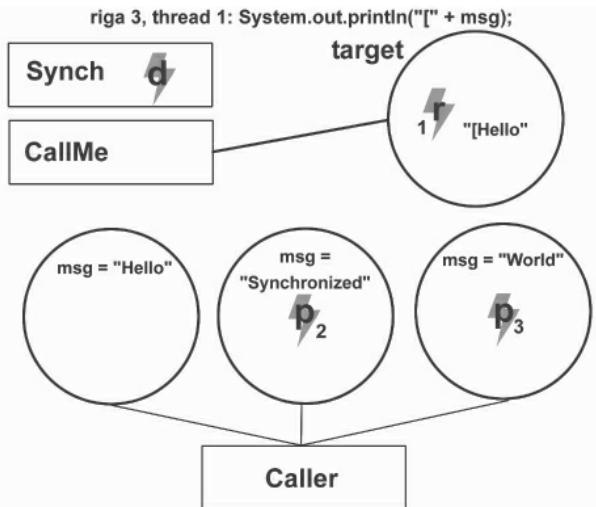


Figura 14.16 - Stampa della prima parentesi e del primo messaggio.

A questo punto il secondo thread si impossessa del processore ripetendo in maniera speculare le azioni del primo thread. Dopo la chiamata al metodo `call()` anche il secondo thread si sposta nel metodo `call()` dell'oggetto `target`, mettendosi a dormire per un secondo (Figura 14.19), dopo aver stampato una parentesi quadra d'apertura e la stringa `Synchronized` (Figura 14.18).

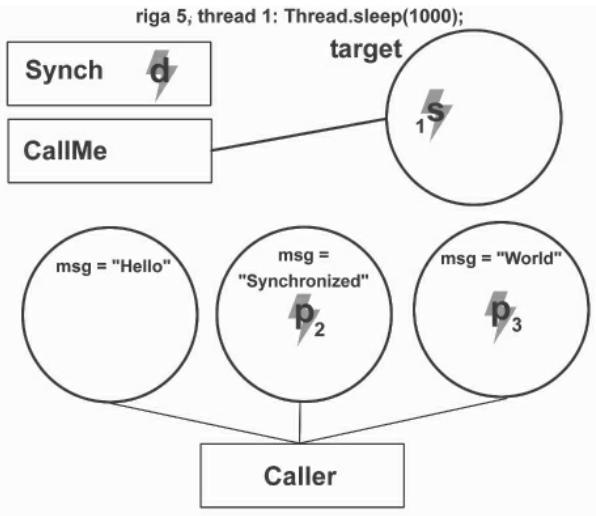


Figura 14.17 - Il thread 1 va a dormire.

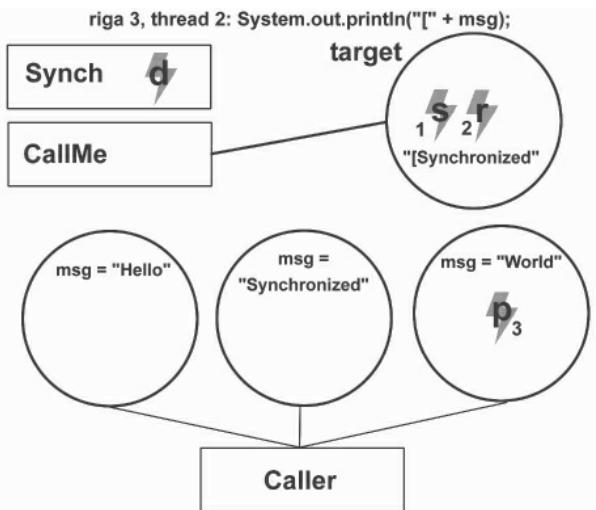


Figura 14.18 - Stampa della seconda parentesi e del secondo messaggio.

Il terzo thread quindi si sposterà nel metodo `call()` dell’oggetto `target`, mettendosi a dormire anch’esso per un secondo (Figura 14.21), dopo aver stampato una parentesi quadra d’apertura e la stringa `World` (Figura 14.20).

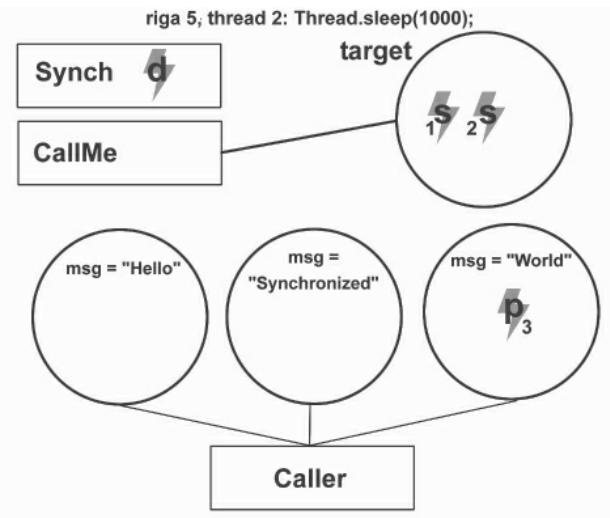


Figura 14.19 - Il thread 2 va a dormire.

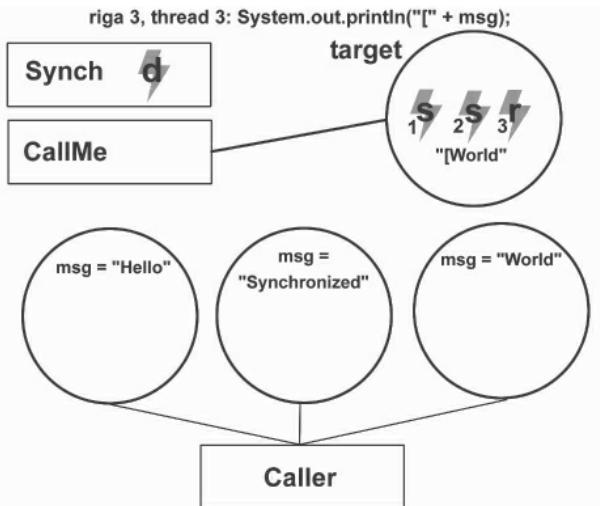


Figura 14.20 - Stampa della terza parentesi e del terzo messaggio.

Dopo poco meno di un secondo si risveglierà il primo thread, che terminerà la sua esecuzione stampando una parentesi quadra di chiusura ed andando a capo (metodo `println()` alla riga 8) (Figura 14.22).

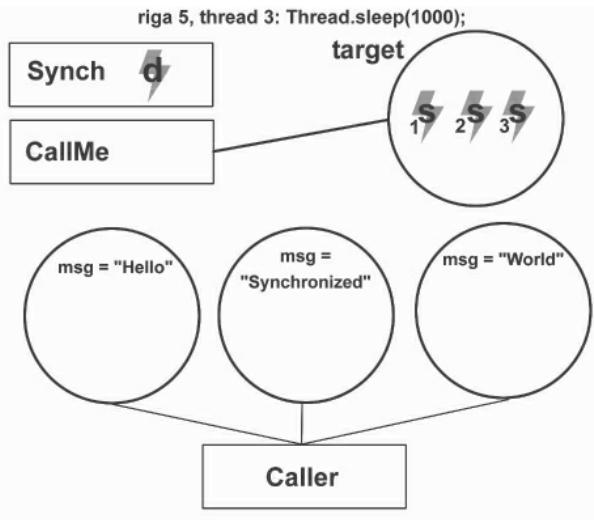


Figura 14.21 - Il thread 3 va a dormire.

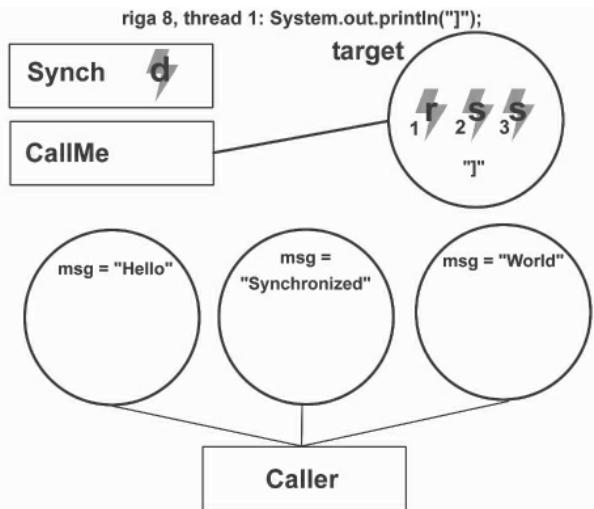


Figura 14.22 - Stampa prima parentesi di chiusura da parte del thread 1.

Anche gli altri due thread si comporteranno allo stesso modo negli attimi successivi (Figura 14.23 e 14.24) producendo l'output non sincronizzato.

L'applicazione ha una durata che si può quantificare in circa un secondo.

Per ottenere l'output sincronizzato basta decommentare il modificatore `synchronized` anteposto alla dichiarazione del metodo `call()` (riga 2). Infatti, quando un thread inizia ad eseguire un metodo

dichiarato sincronizzato, anche in caso di chiamata al metodo `sleep()` non lascia il codice a disposizione di altri thread. Quindi, sino a quando non termina l'esecuzione del metodo sincronizzato, il secondo thread non può eseguire il codice dello stesso metodo. Lo stesso discorso vale quando il secondo ed il terzo thread staranno eseguendo il metodo `synchronized`. L'applicazione quindi ha una durata quantificabile in circa tre secondi e produce un output sincronizzato.

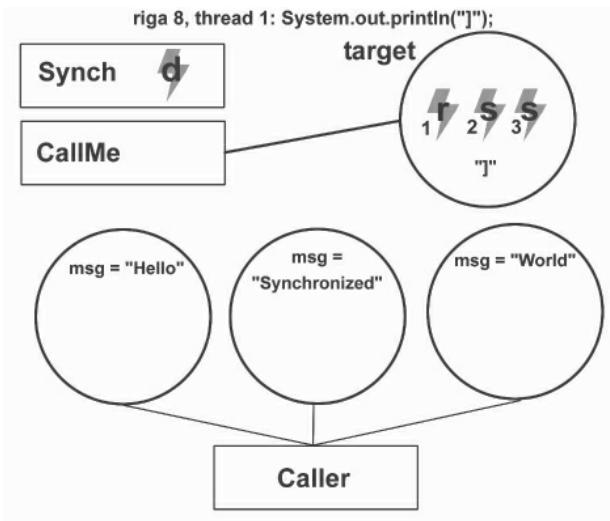


Figura 14.23 - Stampa seconda parentesi di chiusura da parte del thread 2.

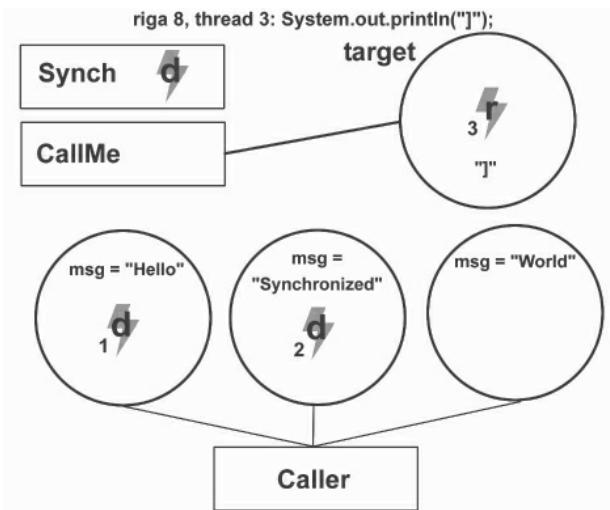


Figura 14.24 - Stampa terza parentesi di chiusura da parte del thread 3.

In alternativa, lo stesso risultato si può ottenere decommentando le righe 20 e 22.

In questo caso la keyword `synchronized` assume il ruolo di comando, tramite la sintassi:

```
synchronized (nomeOggetto) {  
    . . . blocco di codice sincronizzato. . .  
}
```

e quando un thread si trova all'interno del blocco di codice valgono le regole di sincronizzazione sopra menzionate.

Si tratta di un modo di sincronizzare gli oggetti che da un certo punto di vista può risultare più flessibile, anche se più complesso e meno chiaro. Infatti è possibile utilizzare un metodo di un oggetto in maniera sincronizzata o meno a seconda del contesto.

14.4.2 Monitor e Lock

Esiste una terminologia ben precisa riguardante la sincronizzazione dei thread. Nei vari testi che riguardano l'argomento viene definito il concetto di monitor di un oggetto. In Java ogni oggetto ha associato il proprio monitor se contiene codice sincronizzato. A livello concettuale un monitor è un oggetto utilizzato come blocco di mutua esclusione per i thread, il che significa che solo un thread può *entrare* in un monitor in un determinato istante.

Java non implementa fisicamente il concetto di monitor di un oggetto, ma questo è facilmente associabile alla parte sincronizzata dell'oggetto stesso. **Ovvero, se un thread t1 entra in un metodo sincronizzato ms1() di un determinato oggetto o1, nessun altro thread potrà entrare in alcun metodo sincronizzato dell'oggetto o1, sino a quando t1 non avrà terminato l'esecuzione del metodo ms1** (ovvero non avrà abbandonato il monitor dell'oggetto). In particolare si dice che il thread `t1` ha il **lock** dell'oggetto `o1` quando è entrato nel suo monitor (parte sincronizzata).

14.5 La comunicazione fra thread

Nell'esempio precedente abbiamo visto come la sincronizzazione di thread che condividono gli stessi dati sia facilmente implementabile. Purtroppo le situazioni che si presenteranno dove bisognerà gestire la sincronizzazione di thread non saranno sempre così semplici. Come vedremo nel prossimo esempio, la sola keyword `synchronized` non sempre è sufficiente a risolvere i problemi di sincronizzazione fra thread. Descriviamo lo scenario del prossimo esempio. Vogliamo creare una semplice applicazione che simuli la situazione economica ideale, dove c'è un produttore che produce un prodotto e un consumatore che lo consuma. In questo modo il produttore non avrà bisogno di un magazzino. Le attività del produttore e del consumatore saranno processate da due thread eseguiti in attività parallele.

```
// Classe Magazzino *****  
1 public class WareHouse {  
2     private int numberOfWorks;  
3     private int idProduct;
```

```
4     public synchronized void put(int idProduct) {
5         this.idProduct = idProduct;
6         numberOfProducts++;
7         printSituation("Produced " + idProduct);
8     }
9     public synchronized int get() {
10        numberOfProducts--;
11        printSituation("Consumed " + idProduct);
12        return idProduct;
13    }
14    private synchronized void printSituation(String msg) {
15        System.out.println(msg + "\n" + numberOfProducts
16        + " Product in Warehouse");
17    }
18 }
19
20 //classe Produttore ****
21
22 public class Producer implements Runnable {
23     private WareHouse wareHouse;
24     public Producer(WareHouse wareHouse) {
25         this.wareHouse = wareHouse;
26         new Thread(this, "Producer").start();
27     }
28     public void run() {
29         for (int i = 1; i <= 10; i++) {
30             wareHouse.put(i);
31         }
32     }
33 }
34
35 //classe Consumatore ****
36 public class Consumer implements Runnable {
37     private WareHouse wareHouse;
38     public Consumer(WareHouse wareHouse) {
39         this.wareHouse = wareHouse;
40         new Thread(this, "Consumer").start();
41     }
42     public void run() {
43         for (int i = 0; i < 10;) {
44             i = wareHouse.get();
45         }
46     }
47 }
48
49
50 //classe del main ****
51 public class IdealEconomy {
52     public static void main(String args[]) {
53         WareHouse wareHouse = new WareHouse();
54         new Producer(wareHouse);
55         new Consumer(wareHouse);
56     }
57 }
```

Di seguito l'output su Sun Solaris 8:

```
solaris% java IdealEconomy
Produced 1
1 Product in Warehouse
Produced 2
2 Product in Warehouse
Produced 3
3 Product in Warehouse
Produced 4
4 Product in Warehouse
Produced 5
5 Product in Warehouse
Produced 6
6 Product in Warehouse
Produced 7
7 Product in Warehouse
Produced 8
8 Product in Warehouse
Produced 9
9 Product in Warehouse
Produced 10
10 Product in Warehouse
Consumed 10
9 Product in Warehouse
```

14.5.1 Analisi di `IdealEconomy`

Visto l'output prodotto dal codice, l'identificatore della classe `IdealEconomy` suona un tantino ironico!

L'output in questione è stato generato su un sistema Unix (comportamento preemptive) ed è l'output obbligato per ogni esecuzione dell'applicazione. Se l'applicazione fosse stata eseguita su Windows l'output sarebbe potuto variare drasticamente da esecuzione a esecuzione. Su Windows... l'output migliore sarebbe proprio quello che è standard su Unix. Il lettore può eseguire l'applicazione più volte su un sistema Windows per conferma.

Come al solito cerchiamo di immaginare il runtime dell'applicazione, supponendo di eseguire l'applicazione su un sistema Unix (caso più semplice). La classe `IdealEconomy` fornisce il metodo `main()`, quindi partiremo dalla riga 53 dove viene istanziato un oggetto `WareHouse` (letteralmente "magazzino").

La scelta dell'identificatore `WareHouse` può non essere condivisa da qualcuno. Abbiamo

deciso di pensare ad un magazzino perché il nostro obiettivo è tenerlo sempre vuoto.

Nelle successive due righe vengono istanziati un oggetto `Producer` ed un oggetto `Consumer`, ai cui costruttori viene passato lo stesso oggetto `WareHouse` (già si intuisce che il magazzino sarà condiviso fra i due thread). Sia il costruttore di `Producer` sia il costruttore di `Consumer`, dopo aver impostato come variabile d'istanza l'istanza comune di `WareHouse`, creano un thread (con nome rispettivamente `Producer` e `Consumer`) e lo fanno partire. Una volta che il thread principale ha eseguito entrambi i costruttori muore e, poiché è stato fatto partire per primo, il thread `Producer` passa in stato di esecuzione all'interno del suo metodo `run()`. Da questo metodo viene chiamato il metodo sincronizzato `put()` sull'oggetto `wareHouse`. Essendo questo metodo sincronizzato ne è garantita l'atomicità dell'esecuzione, ma nel contesto corrente ciò non basta a garantire un corretto comportamento dell'applicazione. Infatti, il thread `Producer` chiamerà il metodo `put()` dieci volte (riga 29) per poi terminare il suo ciclo di vita e lasciare l'esecuzione al thread `Consumer`. Questo eseguirà un'unica volta il metodo `get()` dell'oggetto `wareHouse`, per poi terminare analogamente il proprio ciclo di vita e con esso il ciclo di vita dell'applicazione.

Sino ad ora non abbiamo visto ancora meccanismi chiari per far comunicare i thread. Il metodo `sleep()`, le priorità e la parola chiave `synchronized` non sono sufficienti. Contrariamente a quanto ci si possa aspettare, questi meccanismi non sono definiti nella classe `Thread` bensì nella classe `Object`.

Trattasi di metodi dichiarati `final` nella classe `Object`, pertanto ereditati da tutte le classi e non modificabili (applicando l'override). Possono essere invocati in un qualsiasi oggetto all'interno di codice sincronizzato.

Questi metodi sono:

- ❑ `wait()`: comunica al thread corrente (quello che legge la chiamata a questo metodo) di abbandonare il monitor e porsi in pausa finché qualche altro thread non entri nello stesso monitor e chiami `notify()`;
- ❑ `notify()`: richiama dallo stato di pausa il primo thread che ha chiamato `wait()` nello stesso oggetto;
- ❑ `notifyAll()`: richiama dalla pausa tutti i thread che hanno chiamato `wait()` in quello stesso oggetto. Viene fra questi eseguito per primo quello a più alta priorità. Quest'ultima affermazione potrebbe non essere verificata da differenti versioni della JVM.

In realtà esistono metodi nella classe `Thread` che realizzano una comunicazione tra thread: `suspend()` e `resume()`. Questi però sono attualmente deprecati, per colpa dell'alta probabilità di produrre deadlock. Il deadlock è una condizione di errore difficile da risolvere, in cui due thread si trovano in una situazione di reciproca dipendenza in due oggetti sincronizzati. Esempio: il thread `t1` si trova nel metodo sincronizzato `m1` dell'oggetto `o1` (di cui quindi possiede il lock) e il thread `t2` si trova nel metodo sincronizzato `m2` dell'oggetto `o2` (di cui quindi possiede il lock). Se il thread `t1` prova a chiamare il metodo `m2` si bloccherà in attesa che il thread `t2` rilasci il lock dell'oggetto `o2`. Se anche il thread `t2` prova a chiamare il metodo `m1` si bloccherà in

attesa che il thread `t1` rilasci il lock dell'oggetto `o1`. L'applicazione rimarrà bloccata in attesa di una interruzione da parte dell'utente. Come si può intuire anche dai soli identificatori, i metodi `suspend()` e `resume()` più che realizzare una comunicazione tra thread di pari dignità, implicavano l'esistenza di thread superiori che avevano il compito di gestirne altri. Poiché si tratta di metodi deprecati, non aggiungeremo altro.

Allo scopo di far comportare correttamente il runtime dell'applicazione `IdealEconomy`, modifichiamo solamente il codice della classe `WareHouse`. In questo modo sarà la stessa istanza di questa classe (ovvero il contesto di esecuzione condiviso dai due thread) a stabilire il comportamento corretto dell'applicazione. In pratica l'oggetto `wareHouse` bloccherà (`wait()`) il thread `Producer` una volta realizzato un `put()` del prodotto, dando via libera al `get()` del `Consumer`. Poi notificherà (`notify()`) l'avvenuta consumazione del prodotto al `Producer` e bloccherà (`wait()`) il `get()` di un prodotto (che non esiste ancora) da parte del thread `Consumer`. Dopo che il `Producer` avrà realizzato un secondo `put()` del prodotto, verrà prima avvertito (`notify()`) il `Consumer` e poi bloccato il `Producer` (`wait()`). Il ciclo si ripete per dieci iterazioni. Per realizzare l'obiettivo si introduce la classica tecnica del flag (`boolean empty` che vale `true` se il magazzino è vuoto). Di seguito vediamo il codice della nuova classe `WareHouse`:

```
1 public class WareHouse {
2     private int numberOfProducts;
3     private int idProduct;
4     private boolean empty = true; // magazzino vuoto
5     public synchronized void put(int idProduct) {
6         if (!empty) // se il magazzino non è vuoto...
7             try {
8                 wait(); // fermati Producer
9             }
10        catch (InterruptedException exc) {
11            exc.printStackTrace();
12        }
13        this.idProduct = idProduct;
14        numberOfProducts++;
15        printSituation("Produced " + idProduct);
16        empty = false;
17        notify(); // svegliati Consumer
18    }
19    public synchronized int get() {
20        if (empty) // se il magazzino è vuoto...
21            try {
22                wait(); // bloccati Consumer
23            }
24            catch (InterruptedException exc) {
25                exc.printStackTrace();
26            }
27        numberOfProducts--;
28        printSituation("Consumed " + idProduct);
29        empty = true; // il magazzino ora è vuoto
30        notify(); // svegliati Producer
31        return idProduct;
32    }
}
```

```

33     private synchronized void printSituation(String msg) {
34         System.out.println(msg +"\n" + numberOfProducts
35             + " Product in Warehouse");
36     }
37 }
```

Immaginiamo il runtime relativo a questo codice. Il thread `Producer` chiamerà il metodo `put()` passandogli 1 alla prima iterazione. Alla riga 6 viene controllato il flag `empty`; siccome il suo valore è `false` non sarà chiamato il metodo `wait()`. Quindi viene impostato l'`idProduct`, incrementato il `numberOfProducts`, e chiamato il metodo `printSituation()`. Il flag `empty` viene impostato a `true` (dato che il magazzino non è più vuoto) e viene invocato il metodo `notify()`. Quest'ultimo non ha alcun effetto dal momento che non ci sono altri thread in attesa nel monitor di questo oggetto. Viene richiamato il metodo `put()` dal thread `Producer` con argomento 2. Questa volta il controllo alla riga 6 viene verificato (`empty` è `false`, cioè il magazzino non è vuoto) e quindi il thread `Producer` chiama il metodo `wait()` rilasciando il lock dell'oggetto. A questo punto il thread `Consumer` esegue il metodo `get()`. Il controllo alla riga 20 fallisce perché `empty` è `false`, quindi non viene chiamato il metodo `wait()`. Viene decrementato `numberOfProducts` a 0 e chiamato il metodo `printSituation()`; si imposta il flag `empty` a `true`. Poi viene chiamato il metodo `notify()`, che toglie il thread `Producer` dallo stato di `wait()`. Viene restituito il valore di `idProduct` che è ancora 1. Poi viene richiamato il metodo `get()`, ma il controllo alla riga 20 è verificato e dunque il thread `Consumer` si mette in stato di `wait()`. Quindi il thread `Producer` continua la sua esecuzione da dove si era bloccato cioè dalla riga 8. Viene impostato `idProduct` a 2, incrementato di nuovo ad 1 `numberOfProducts` e chiamato il metodo `printSituation()`. Il flag `empty` viene impostato a `false` e viene chiamato il metodo `notify()`, che risveglia il thread `Consumer`.

14.6 Concorrenza

Quando abbiamo a che fare con concetti tanto complicati, è sconsigliato re-inventarsi ogni volta l'acqua calda. Benché a volte ci sia la convinzione di avere la situazione sotto controllo con i thread non si è mai troppo sicuri. Inoltre i bachi creati dall'utilizzo dei thread sono molto difficili da risolvere e spesso sono anche molto dannosi!

Come abbiamo già detto, la gestione dei thread introduce una nuova dimensione alla programmazione: la dimensione temporale. La *programmazione multi-threaded* è da considerarsi un nuovo tipo di programmazione. Non è la tipica programmazione fatta solo di `if` e `for`, ed esistono interi libri che parlano solo di questo argomento. Durante la lettura di queste pagine abbiamo già parlato di altri tipi di programmazione: prima della *programmazione object oriented*, poi della *programmazione per contratto*, poi della *programmazione generica*, e quando parleremo di espressioni lambda nel prossimo modulo, approcceremo ad un altro tipo di programmazione ancora. Ogni volta è un po' come partire da zero e la cosa migliore da fare è sfruttare le tecniche e i design pattern conosciuti. Per esempio nel codice di `IdealEconomy` abbiamo usato una tecnica nota come *guarded block* per gestire la comunicazione nella classe `Warehouse` tra `Producer` e `Consumer`. È grazie ad essa che abbiamo pensato a chiamare il metodo `wait()` in un ciclo, e far dipendere la chiamata a `notify()` da un booleano, non ci siamo inventati niente. Proprio per questo motivo in

Java esistono librerie che mettono a disposizione implementazioni pronte per l'uso, che risolvono la maggior parte dei tipici problemi che si presentano nella programmazione multi-threaded. Tuttavia senza conoscere l'argomento è difficile utilizzare questi strumenti in maniera consapevole.

14.6.1 Oggetti Immutabili

Durante lo studio di questo libro abbiamo incontrato diverse classi che abbiamo definito immutabili. Con questo aggettivo abbiamo inteso che gli oggetti istanziati da queste classi non potevano cambiare il loro stato interno. È il caso di `Wrapper`, `Math`, `String` e di tutte le classi della libreria “`Date-Time API`”. Il vantaggio di usare oggetti immutabili in luogo degli oggetti mutabili, è che gli oggetti immutabili sono thread-safe (in italiano sicuri per l'utilizzo dei thread). È un concetto semplice da comprendere dopo quello che abbiamo visto nei precedenti paragrafi: un oggetto thread-safe non corre il rischio di essere modificato internamente da più thread concorrenti contemporaneamente. Per esempio due thread `t1` e `t2` potrebbero accedere contemporaneamente all'oggetto `o1` che contiene una variabile `value` che al momento della lettura dei due thread vale `0`. Poi entrambi provano a modificare la variabile con lo scopo di incrementarla di un'unità. Supponiamo che il thread `t1` arrivi prima e modifichi il valore di `value` a `1`. Quando arriverà il thread `t2` incrementerà il valore di `value` a `2`, ma l'obiettivo di `t2` era quello di incrementare il valore ad `1` dal momento che al momento della lettura `value` valeva `1`. Ovviamente `o1` non era un oggetto thread-safe.

Un oggetto immutabile invece è thread-safe visto che il suo stato interno non è modificabile. Inoltre gli oggetti immutabili garantiscono altri vantaggi relativi al loro trattamento in memoria da parte della virtual machine. Le linee guida per creare una classe immutabile sono le seguenti:

1. tutti gli attributi devono essere dichiarati `final` e `private`.
2. Non fornire alla classe metodi di tipo “set”, o che modifichino le variabili d’istanza in generale.
3. Nel caso i valori delle variabili d’istanza di tipo reference debbano essere passati in input a metodi dall'esterno, bisogna preoccuparsi di creare delle copie da immagazzinare internamente, per evitare che siano modificate dall'esterno successivamente.
4. Nel caso variabili interne di tipo reference debbano essere esposte tramite metodi “get”, bisogna preoccuparsi di creare delle copie da restituire all'esterno, per evitare che siano modificate dall'esterno successivamente.
5. Impedire alle sottoclassi di fare override dei metodi. Quindi dichiarare `final` la classe dovrebbe essere sufficiente. Dichiарare `final` anche i metodi metterebbe ancora più in risalto le nostre intenzioni.

Se vogliamo modificare un oggetto immutabile dobbiamo creare dei metodi che restituiscano altri oggetti modificabili a partire dall'originale. Un buon esempio sono i metodi di `String` come `toUpperCase()`. Il codice:

```
String nome = "Simone";
String nomeMaiuscolo = nome.toUpperCase();
System.out.println(nome);
```

```
System.out.println(nomeMaiuscolo);
```

stamperà:

```
Simone  
SIMONE
```

in quanto la variabile `nome` non è cambiata. Di seguito un esempio di classe immutabile e quindi thread-safe:

```
import java.util.*;  
  
public class ImmutableObject {  
    private final int integer; //variabile d'istanza primitiva  
    private final String string; //variabile d'istanza complessa immutabile  
    private final Calendar calendar; //variabile d'istanza complessa  
    // Costruttore che prende le variabili dall'esterno  
    public ImmutableObject (int integer, String string, Calendar calendar) {  
        this.integer = integer;  
        this.string = string;  
        // segue punto 3  
        this.calendar = (Calendar)calendar.clone();  
    }  
    //Questo metodo segue il punto 4  
    public Calendar getDate() {  
        return (Calendar)calendar.clone();  
    }  
    //Metodo di test  
    public void stampaCalendar() {  
        System.out.println(calendar);  
    }  
}
```

14.6.2 Tecnologie Java e classi standard ausiliarie

La gestione dei thread in Java può considerarsi semplice se paragonata alla gestione dei thread in altri linguaggi. Inoltre una classe (`Thread`), un'interfaccia (`Runnable`), una parola chiave (`synchronized`) e tre metodi (`wait()`, `notify()` e `notifyAll()`) rappresentano il nucleo di conoscenza fondamentale per gestire applicazioni multithreaded. In realtà le situazioni multithreaded che bisognerà gestire nelle applicazioni reali non saranno tutte così semplici come nell'ultimo esempio. Tuttavia la tecnologia Java fa ampio uso del multithreading. La notizia positiva è che la complessità del multithreading viene spesso gestita dalla tecnologia stessa. Consideriamo per esempio la tecnologia Web Java Servlet (<http://www.oracle.com/technetwork/java/javaee/servlet/index.html>).

Java Servlet è in grado di gestire il multithreading in modo automatico, allocando thread diversi per servire ogni richiesta del client. Quindi lo sviluppatore è completamente esonerato dal dover scrivere codice per la gestione dei thread. Gestiscono spesso thread automaticamente anche altre

tecnologie Java Web (per esempio Java Server Pages), Enterprise (per esempio Enterprise JavaBeans) e Standard (per esempio le interfacce grafiche). Poi ci sono alcune classi della libreria Java (`java.nio.channels.SocketChannel`, `java.util.Timer`, `java.util.concurrent.ThreadLocalRandom` etc.) che semplificano enormemente il rapporto tra lo sviluppatore ed i thread. Per esempio prendiamo la classe `TimerTask`. Essa ci consente di creare un semplice thread che si deve risvegliare dopo un certo lasso di tempo, per eseguire un determinato compito. Questa classe viene utilizzata da EJE per creare la funzionalità di sveglia. In particolare bisogna fare due semplici cose:

1. creare una sottoclasse della classe astratta `TimerTask` (che a sua volta implementa l'interfaccia `Runnable`) che definisce un metodo `run()` che sarà chiamato al risveglio del `Timer`.
2. Schedulare l'evento passando un oggetto `TimerTask` al metodo `schedule()` di un oggetto `Timer`.

Riguardo il passo 2, ecco come EJE schedula un evento sveglia (codice modificato per semplicità):

```
new      java.util.Timer().schedule(new           AlarmClockTimerTask(messaggio),  
millisecondi);
```

dove `AlarmClockTimerTask` è l'implementazione del `TimerTask`, `messaggio` è una stringa da stampare e `millisecondi` è il numero di millisecondi che devono passare affinché il `TimerTask` sia eseguito. Semplicissimo e pulito. È possibile anche schedulare anche più di un `TimerTask` ed esistono diversi overload del metodo `schedule()` della classe `Timer` per schedulare i `TimerTask` ad intervalli regolari, usando un oggetto `Date` oppure un ritardo di tempo in millisecondi di tipo `long` (cfr. documentazione `java.util.Timer`).

Esistono anche una serie di classi ed interfacce del framework “Collections” che servono proprio per essere utilizzate in ambienti multi threading. Le varie `Queue`, `BlockingQueue` e `ConcurrentMap` verranno affrontate però nel modulo 16 dedicato alle collezioni.

14.6.3 Libreria Java sulla concorrenza

Dalla versione 5 di Java, il supporto al multithreading è stato ulteriormente ampliato con l'introduzione di classi molto potenti che semplificano il lavoro dello sviluppatore. Con il package `java.util.concurrent` e i suoi sotto-package Java definisce una serie di strumenti di alto livello per gestire la concorrenza fra thread in maniera semplice ed elegante.

14.6.3.1 Il package `java.util.concurrent.locks`

Il package `java.util.concurrent.locks` definisce una serie di classi ed interfacce che permettono di gestire in maniera più semplice i lock, rispetto ai vari `synchronized`, `wait()` e `notify()` che abbiamo visto negli esempi precedenti. L'interfaccia principale si chiama `Lock`, e definisce tra i vari metodi due versioni (overload) del metodo `tryLock()`. Una di queste prende in input dei valori per specificare un time-out affinché un thread esca dall'impasse nel caso non si riesca a prendere il lock

di un oggetto. La più famosa delle implementazioni di `Lock` si chiama `ReentrantLock`, e nel seguente esempio si può facilmente comprendere come funziona:

```
import java.util.concurrent.locks.*;
import java.util.concurrent.*;

public class LockExample implements Runnable {
    private int numeroIntero;
    private Lock lock;

    public LockExample(int numeroIntero) {
        this.numeroIntero = numeroIntero;
        this.lock = new ReentrantLock();
    }

    @Override
    public void run() {
        try {
            if(lock.tryLock(5, TimeUnit.SECONDS)) {
                numeroIntero++;
                System.out.println("In lock "+numeroIntero);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
        System.out.println("Out lock "+numeroIntero);
    }
}
```

All'interno di questa classe che implementa `Runnable` (e quindi codice eseguibile per thread), nel metodo `run()`, un oggetto `lock` prova a prendere il lock dell'oggetto corrente, con un time-out di cinque secondi. Se vi riesce entra nel blocco di codice dell'`if`, prende il lock dell'oggetto e lo rilascia nella clausola `finally`.

A differenza della parola chiave `synchronized`, un oggetto `Lock` ha quindi la possibilità di impostare un tempo di time-out, e di specificare un `unlock()` nel codice di `finally`. Un altro punto a favore dell'utilizzo di un `ReentrantLock` rispetto ad un blocco sincronizzato, è quello di poter creare lock con “fairness”, che in italiano potremo tradurre con “correttezza”. Stiamo parlando di un semplice booleano che può essere passato in input al costruttore di `ReentrantLock`, denominato `fair`. Se impostato a `true`, questo booleano consentirà allo scheduler della JVM di creare una gerarchia di priorità di esecuzione tra thread, basata sul tempo di creazione del `ReentrantLock`. Prima è stato creato l'oggetto `lock`, e maggiore è la sua priorità. Quindi, nel caso in un certo istante sia rilasciato il lock di un oggetto, e diversi thread abbiano la possibilità di prendere questo lock, sarà favorito il più vecchio. Con la sincronizzazione ordinaria non è possibile creare tale gerarchia.

Uno svantaggio del `ReentrantLock` invece, può essere considerato proprio l'obbligo di dover utilizzare un blocco `try-catch-finally` (con il catch facoltativo), il che rende il codice pesante e

verboso. Fortunatamente gli IDE ci vengono incontro con la creazione automatica dei blocchi `try-catch-finally`.

Nello stesso package esistono anche altri interessanti tipi come l'interfaccia `Condition` e `StampedLock`. Mentre possiamo considerare `Lock` come un'alternativa ad un blocco o un metodo `synchronized`, `Condition` può essere considerata come l'alternativa all'utilizzo dei metodi per la gestione del monitor di un oggetto: `wait()`, `notify()` e `notifyAll()`. Si ottiene un'istanza di `Condition` tramite il metodo `newCondition()` chiamato su un `Lock`, e si usa il metodo `await()` per sostituire `wait()`, e i metodi `signal()` e `signalAll()` per sostituire rispettivamente `notify()` e `notifyAll()`. Esistono anche diversi overload di `await()` tra cui un paio che permettono di specificare un time-out.

La classe `StampedLock` è invece una novità di Java 8. Si tratta di una nuova tipologia di lock, che consente il lock di un oggetto mediante tre modi: "scrittura", "lettura" e "lettura ottimistica" basata su timestamp, ovvero degli istanti calcolati come `long` che serviranno per validare le varie azioni sull'oggetto.

14.6.3.2 Il package `java.util.concurrent.atomic`

Il package `java.util.concurrent.atomic` definisce una serie di classi come `AtomicLong`, `AtomicDouble` etc., che permettono di effettuare operazioni atomiche in maniera thread-safe. I metodi di tali classi combinano più operazioni come `getAndDecrement()`, `getAndIncrement()`, `compareAndSet()` e `getAndSet()`, che garantiscono l'atomicità dell'esecuzione di queste operazioni, anche nel caso più thread accedessero contemporaneamente alla stessa variabile. Per esempio alla fine del paragrafo 14.3.4, abbiamo evidenziato il fatto che le operazioni di pre e post incremento (o decremento) non sono operazioni atomiche e quindi non thread-safe. Con le classi di questo package possiamo risolvere questo problema in maniera semplice ed elegante. Per esempio possiamo chiamare il metodo `incrementAndGet()` in luogo dell'operatore di pre-incremento:

```
AtomicLong atomicLong = new AtomicLong(10);
long incrementedLong = atomicLong.incrementAndGet();
```

Altre classi appartenenti a questo package sono gli **adder** (`LongAdder`, `DoubleAdder`) e gli **accumulator** (`LongAccumulator`, `DoubleAccumulator`). Entrambi i tipi di classi consentono di fare operazioni atomiche su più elementi. Ma si tratta di classi che apprezzeremo meglio dopo aver studiato le espressioni lambda.

14.6.3.3 Interfacce Executors

Nel package `java.util.concurrent` sono definite tre interfacce note come **Executors**. Si tratta di semplici interfacce che definiscono un metodo `execute()`, il quale permette di eseguire thread in maniera molto semplice. Un metodo `execute()` prende in input un oggetto `Runnable` e viene invocato su un oggetto executor. Ricordiamo invece che abbiamo imparato a creare thread, al cui costruttore passavamo un oggetto `Runnable` e sul thread chiamavamo il metodo `start()`. La grande

differenza rispetto a quanto visto sino ad ora, è che gli “executors” permettono di separare nettamente il codice dei thread e degli oggetti `Runnable`. Dal punto di vista architettonale questa scelta è più che opportuna. Questo consente di gestire i nostri programmi con più semplicità, senza mischiare codice funzionale con codice per gestire la concorrenza.

Le interfacce executors sono:

1. `Executor`: una semplice interfaccia che permette l'esecuzione di thread.
2. `ExecutorService`: un'interfaccia che estende `Executor` e che definisce anche un metodo `submit()`, più completo rispetto ad `execute()`. Infatti il `submit()` può prendere in input un oggetto di tipo `Callable` in luogo di un oggetto `Runnable`, e restituisce un oggetto di tipo `Future`, che permette di recuperare dati in output dopo l'esecuzione del metodo. `ExecutorService` inoltre aggiunge funzionalità per gestire il ciclo di vita sia dei processi che dell'executor stesso.
3. `ScheduledExecutorService`: estende `ExecutorService` e consente di schedulare i processi.

Probabilmente l'esempio più famoso di executor è la creazione di un pool di thread, da sempre una delle implementazioni più complicate da creare per un programmatore Java. Invece in Java gli oggetti `ExecutorService` permettono con poche righe di codice la creazione di pool di thread. Il pattern dei “threads pool”, nella sua definizione più banale, permette di creare una pool, ovvero una riserva, dove un numero fisso di thread già pronti aspetta di essere usato. Quando c'è bisogno di eseguire un processo, viene richiesta la disponibilità all'executor, che restituisce la prima istanza di thread disponibile (ovvero che non stia già lavorando). Quando un thread finisce il suo compito ritorna nella pool in attesa di essere usato. Questo pattern consente di limitare il numero di istanze dei thread, e fa sì che si possa evitare la creazione di un thread a richiesta, visto che è un processo dispendioso per le risorse del programma. Inoltre un'implementazione efficiente permette performance migliori. Segue un semplice esempio di implementazione di `ExecutorService`:

```
import java.util.concurrent.*;  
  
public class TestExecutorService {  
    public static void main(String args[]) throws Exception {  
        ExecutorService service = Executors.newFixedThreadPool(10);  
        for (int i = 0; i < 100; i++) {  
            Future<Integer> future = service.submit(new Processo(i));  
            System.out.println(" Valore: " + future.get());  
        }  
        service.shutdownNow();  
    }  
  
    class Processo implements Callable<Integer> {  
        private int id;  
  
        public Processo(int id) {  
            this.id = id;  
        }  
    }  
}
```

```

@Override
public synchronized Integer call() {
    System.out.print("ID : " + this.id + ", thread : "
        + Thread.currentThread().getName());
    return this.id;
}
}

```

Con una classe interna abbiamo creato un'implementazione dell'interfaccia generica `Callable`, che nel suo metodo `call()` (che equivale al metodo `run()` di `Runnable`) stampa la situazione e restituisce un `Integer`, definito direttamente come tipo parametro di `callable`. Possiamo invece osservare nel metodo `main()` della classe `TestExecutorService`, che viene istanziato un oggetto `ExecutorService` mediante il metodo `newFixedThreadPool()`, che istanzia una pool non ridimensionabile di 10 elementi. Poi all'interno del ciclo `for` viene invocato il metodo `submit()` (che equivale al vecchio metodo `start()` della classe `Thread`) sull'oggetto `ExecutorService`. Viene poi catturato l'oggetto `Future` con tipo parametro `Integer`, che contiene il risultato dell'operazione.

È possibile creare pool di thread più complessi con `ThreadPoolExecutor` e `ScheduledThreadPoolExecutor` (cfr. documentazione ufficiale).

Con Java 7 è stato anche introdotto un nuovo framework detto “Fork/Join” che estende in qualche modo le funzionalità degli executors, mettendo a disposizione dell'utente un algoritmo basato sulla ricorsione detto “work-stealing”. Con l'utilizzo di questo framework i thread del pool che hanno completato il loro compito, possono “rubare” ad altri thread ancora impegnati parti di codice da eseguire. Questo permette di ottimizzare i tempi di esecuzione in ambienti multi-processori. Il framework si basa sulla classe `ForkJoinPool` (un'estensione della classe astratta `AbstractExecutorService`) e il concetto di dividere il codice da eseguire in più parti mediante l'implementazione di una tra le classi astratte `RecursiveAction` o `RecursiveTask`. Quest'ultima ha la possibilità di conservare un valore di ritorno per l'operazione da eseguire. Una tipica implementazione del framework Fork/ Join, è il metodo `parallelSort()` della classe `Arrays`, che ordina gli elementi di un array. L'array viene suddiviso in sotto array che vengono ordinati in parallelo per poi essere a loro volta ricongiunti (join) per ottenere il risultato finale.

14.6.3.4 La classe `Semaphore`

Ci sono altre classi che sono degne di nota. Per esempio in questa libreria esiste il concetto di semaforo, argomento già familiare a chi conosce la gestione dei thread in altri ambienti. Gli oggetti della classe `Semaphore` concettualmente funzionano in maniera molto similare a come funzionano i veri semafori. Quando è rosso (ovvero il semaforo ha acquisito il lock sull'oggetto) non si passa, quando è verde si passa (ovvero i thread possono prendere il lock dell'oggetto). La classe `Semaphore` mette a disposizione metodi come `acquire()` e `release()` che garantiscono il lock e l'unlock del monitor di un oggetto. Viene anche definito il metodo `tryAcquire()` che consente di specificare un timeout massimo di attesa per prendere il lock. Come abbiamo visto anche con gli oggetti `Lock`, la gestione di un semaforo si fa preferire ai comandi “nudi e crudi” `wait()` e

`notify()`. In particolare i semafori permettono di restringere a un certo numero di thread l'accesso contemporaneo ad una risorsa. Nel seguente esempio creiamo come prima cosa una classe `Veicolo` che implementa `Runnable`:

```
import java.util.concurrent.Semaphore;

public class Veicolo implements Runnable {
    private Incrocio incrocio;
    public Veicolo(Incrocio incrocio) {
        this.incrocio = incrocio;
    }

    public void run() {
        incrocio.rispettaSemaforo();
    }
}
```

Questa definisce un'aggregazione con la classe seguente `Incrocio`, che imposta come variabile d'istanza tramite il suo costruttore. Il suo metodo `run()` chiama il metodo `rispettaSemaforo()` sull'oggetto `incrocio`:

```
import java.util.concurrent.Semaphore;

public class Incrocio {
    Semaphore semaforo = new Semaphore(1);

    public void rispettaSemaforo() {
        try {
            semaforo.acquire();
            System.out.println(Thread.currentThread().getName()
                + " sta superando l'incrocio");
            Thread.sleep(1000);
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        } finally {
            semaforo.release();
            System.out.println(Thread.currentThread().getName()
                + " ha superato l'incrocio");
        }
    }
}
```

La classe `Incrocio` definisce una variabile d'istanza di tipo `Semaphore`, al cui costruttore è stato passato il valore `1` che rappresenta il numero di “permits” (permessi) che possono essere concessi. `permits` non è altro che un contatore che mantiene l'oggetto `semaforo` per concedere l'accesso alla risorsa condivisa. In questo esempio un solo thread alla volta potrà superare il metodo `acquire()` senza che un altro thread chiami il metodo `release()` sullo stesso oggetto `semaforo`. Le seguenti righe di codice:

```
Incrocio incrocio = new Incrocio();
Thread t1 = new Thread(new Veicolo(incrocio), "Una Ford");
t1.start();
Thread t2 = new Thread(new Veicolo(incrocio), "Una Nissan");
t2.start();
```

consentiranno di simulare la situazione dove solo una macchina per volta supererà l'incrocio. Segue l'output:

```
Una Ford sta superando l'incrocio
Una Ford ha superato l'incrocio
Una Nissan sta superando l'incrocio
Una Nissan ha superato l'incrocio
```

14.6.3.5 La classe CyclicBarrier

Vale la pena dare uno sguardo anche alla classe `CyclicBarrier`. Questa fornisce la possibilità ad un gruppo di thread di sincronizzarsi e attendersi l'un l'altro in un determinato punto del codice (da qui il nome della classe). In seguito viene presentato un esempio che simula la situazione in cui dei partecipanti ad una festa devono essere tutti presenti affinché la festa abbia inizio. Prima definiamo la classe `Festa` che implementa `Runnable` e definisce come variabile d'istanza una `CyclicBarrier`. Questa farà da punto di sincronizzazione all'interno del metodo `run()` fra più thread. Quando un thread arriva a chiamare il metodo `await()`, si blocca ad aspettare che altri thread arrivino allo stesso punto per poter continuare la sua esecuzione:

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class Festa implements Runnable {
    private CyclicBarrier luogoDellaFesta;

    public Festa(CyclicBarrier luogoDellaFesta) {
        this.luogoDellaFesta = luogoDellaFesta;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName()
                + " è presente");
            luogoDellaFesta.await();
            System.out.println(Thread.currentThread().getName()
                + " ha iniziato a festeggiare...");
        } catch (InterruptedException | BrokenBarrierException ex) {
            ex.printStackTrace();
        }
    }
}
```

```
}
```

La classe seguente crea una `CyclicBarrier` passando al suo costruttore il numero dei thread che devono chiamare `await()` per sbloccare la barriera, e un oggetto `Runnable` creato al volo sotto forma di classe anonima che definisce l'azione che deve essere eseguita dalla `CyclicBarrier` una volta sbloccata. Inoltre definisce i protagonisti della festa (i thread), e li fa partire:

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class TestFesta {
    public static void main(String args[]) {
        CyclicBarrier luogoDellaFesta = new CyclicBarrier(3, new Runnable() {
            @Override
            public void run() {
                System.out.println("Tutti presenti, possiamo iniziare!");
            }
        });
        Festa festa = new Festa(cb);
        new Thread(festa, "Antonio").start();
        new Thread(festa, "Marcello").start();
        new Thread(festa, "Serena").start();
    }
}
```

L'output di quest'applicazione è il seguente:

```
Marcello è presente
Serena è presente
Antonio è presente
Tutti presenti, possiamo iniziare!
Antonio ha iniziato a festeggiare...
Marcello ha iniziato a festeggiare...
Serena ha iniziato a festeggiare...
```

ma l'ordine di arrivo dei partecipanti alla festa potrà cambiare ad ogni esecuzione.

Riepilogo

Abbiamo distinto le definizioni di **multithreading** e **multitasking**. Abbiamo gradualmente dato la definizione di **thread**, che è piuttosto complessa (processore virtuale che esegue codice su determinati dati). Abbiamo utilizzato la classe `Thread` e l'interfaccia `Runnable` per avere codice da far eseguire ai nuovi thread. Abbiamo parlato di **scheduler** e **priorità** ed abbiamo visto come questo concetto non sia quello determinante per gestire più thread contemporaneamente. È stato introdotto il modificatore `volatile`, che è applicabile solo alle variabili per indicare alla virtual machine di non

creare copie di esse in memoria a runtime. Abbiamo imparato a **sincronizzare** più thread introducendo i concetti di **monitor** e **lock** di un oggetto in due modi diversi: sincronizzando un metodo oppure un'istanza in un determinato blocco di codice. Abbiamo visto che è spesso necessario sincronizzare alcune parti di codice quando questo è eseguibile da più thread. La sincronizzazione si limita a sincronizzare il codice, non i dati utilizzati. Abbiamo anche esplorato con un esempio la **comunicazione tra thread**, che si gestisce direttamente dal codice da eseguire in comune mediante i metodi `wait()`, `notify()` e `notifyAll()` della classe `Object`. Infine abbiamo accennato alle librerie e le tecnologie legate ai thread, ed in particolare al package `java.util.concurrent` ed ai suoi sottopackage che introducono strutture di alto livello per la **gestione avanzata dei thread**.

Esercizi modulo 14

Esercizio 14.a) Creazione di thread, Vero o Falso:

1. Un thread è un oggetto istanziato dalla classe `Thread` o dalla classe `Runnable`.
2. Il multithreading è solitamente una caratteristica dei sistemi operativi e non dei linguaggi di programmazione.
3. In ogni applicazione al runtime esiste almeno un thread in esecuzione.
4. A parte il thread principale, un thread ha bisogno di eseguire codice all'interno di un oggetto la cui classe estende `Runnable` o estende `Thread`.
5. Il metodo `run()` deve essere chiamato dal programmatore per attivare un thread.
6. Il thread corrente non si identifica solitamente con il reference `this`.
7. Chiamando il metodo `start()` su di un thread, questo viene immediatamente eseguito.
8. Il metodo `sleep()` è statico e permette di far dormire per un numero specificato di millisecondi il thread che legge tale istruzione.
9. Assegnare le priorità ai thread è una attività che può produrre risultati diversi su piattaforme diverse.
10. Lo scheduler della JVM non dipende dalla piattaforma su cui viene eseguito.

Esercizio 14.b) Gestione del multi-threading, Vero o Falso:

1. Un thread astrae un processore virtuale che esegue codice su determinati dati.
2. La parola chiave `synchronized` può essere utilizzata sia come modificatore di un metodo sia come modificatore di una variabile.
3. Il monitor di un oggetto può essere identificato con la parte sincronizzata dell'oggetto stesso.
4. Affinché due thread che eseguono lo stesso codice e condividono gli stessi dati non abbiano problemi di concorrenza, è necessario sincronizzare il codice comune.

5. Si dice che un thread ha il lock di un oggetto se entra nel suo monitor.
6. I metodi `wait()`, `notify()` e `notifyAll()` rappresentano il principale strumento per far comunicare più thread.
7. I metodi `suspend()` e `resume()` sono attualmente deprecati.
8. Il metodo `notifyAll()`, invocato su di un certo oggetto `o1`, risveglia dallo stato di pausa tutti i thread che hanno invocato `wait()` sullo stesso oggetto. Tra questi verrà eseguito quello che era stato fatto partire per primo con il metodo `start()`.
9. Il deadlock è una condizione di errore bloccante generata da due thread che stanno in reciproca dipendenza in due oggetti sincronizzati.
10. Se un thread `t1` esegue il metodo `run()` nell'oggetto `o1` della classe `C1`, e un thread `t2` esegue il metodo `run()` nell'oggetto `o2` della stessa classe `C1`, la parola chiave `synchronized` non serve a niente.

Soluzioni esercizi modulo 14

Esercizio 14.a) Creazione di `thread`, Vero o Falso:

1. **Falso.** `Runnable` è un'interfaccia.
2. **Vero.**
3. **Vero**, il cosiddetto thread “main”.
4. **Vero.**
5. **Falso**, il programmatore può invocare il metodo `start()` e lo scheduler invocherà il metodo `run()`.
6. **Vero.**
7. **Falso.**
8. **Vero.**
9. **Vero.**
10. **Falso.**

Esercizio 14.b) Gestione del multithreading, Vero o Falso:

1. **Vero.**
2. **Falso.**
3. **Vero.**
4. **Falso.**

5. Vero.

6. Vero.

7. Vero.

8. Falso, il primo thread che partirà sarà quello a priorità più alta.

9. Vero.

10. Vero.

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi

<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper definire multithreading e multitasking (unità 14.1)	<input type="checkbox"/>	
Comprendere la dimensione temporale introdotta dalla definizione dei thread in quanto oggetti (unità 14.2)	<input type="checkbox"/>	
Saper creare ed utilizzare thread tramite la classe <code>Thread</code> e l'interfaccia <code>Runnable</code> (unità 14.2)	<input type="checkbox"/>	
Definire che cos'è uno scheduler e i suoi comportamenti riguardo alle priorità dei thread (unità 14.3)	<input type="checkbox"/>	
Saper utilizzare i modificatori <code>volatile</code> e <code>synchronized</code> (unità 14.3, 14.4)	<input type="checkbox"/>	
Far comunicare i thread (unità 14.5)	<input type="checkbox"/>	
Conoscere il nucleo delle principali librerie sulla concorrenza (unità 14.6)	<input type="checkbox"/>	

Note:

Espressioni Lambda

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Comprendere la sintassi e le modalità d'uso che ci offre la nuova feature delle espressioni lambda (unità 15.1).
- ✓ Saper decidere quando utilizzare un'espressione lambda o una classe anonima (unità 15.1).
- ✓ Saper gestire la dinamicità del passaggio di codice ad un metodo con un'espressione lambda (unità 15.1).
- ✓ Saper definire ed utilizzare tutte le quattro diverse sintassi dei reference a metodi (unità 15.2).
- ✓ Saper utilizzare le interfacce funzionali definite nel package `java.util.function` (unità 5.3).

Quando Java nacque nel 1995, era un linguaggio molto più semplice del linguaggio che stiamo studiando in queste pagine. Non esistevano generici, classi anonime, cicli `foreach`, import statici, `varargs`, enumerazioni, asserzioni, annotazioni e tante altre librerie. Negli anni sono stati introdotte gradualmente queste caratteristiche ad una ad una, e il linguaggio è diventato sempre più potente e completo, ma anche più complicato da imparare. Soprattutto nella versione 5, Java subì una clamorosa svolta con l'introduzione di tantissime novità. Quella che però non era mai cambiata sino ad ora, era la coerenza d'essere un linguaggio orientato agli oggetti. Ogni nuova funzionalità che veniva introdotta era sempre progettata per integrarsi nel modello di programmazione orientato agli oggetti. In diciannove anni si è sempre cercato di far rimanere Java un linguaggio "più ad oggetti possibile".

Questo ha fatto di Java il linguaggio di riferimento per il paradigma Object Oriented. Tanto per fare un esempio, oggi negli atenei italiani si insegna Java (in molti casi usando questo libro) anche per spiegare gli oggetti. Nella didattica accademica altri linguaggi object oriented sono stati accantonati nel tempo, perché il supporto ai paradigmi non è così chiaro e definito come quello che offre Java. La coerenza di Java è sempre stata apprezzata, e la filosofia dell'Object Orientation si è affermata nel tempo. Java è considerato da molti il linguaggio ideale per progettare soluzioni ad oggetti, anche se lo scotto da pagare era una certa prolissità nel codice degli algoritmi.

Negli ultimi tempi però la scena della programmazione mondiale è cambiata. Con l'avvento di processori multi-core nell'uso domestico, la programmazione funzionale è stata rivalutata. Questo è successo un po' perché essa ben si adatta alle nuove architetture hardware, e un po' perché il suo accantonamento in favore dell'Object Orientation, fu dovuto soprattutto al suo utilizzo improprio. In fondo il paradigma funzionale ha dominato la scena della programmazione per decenni prima di passare in secondo piano negli anni novanta. Grazie anche alle novità portata da linguaggi moderni come Scala, Groovy e il framework Microsoft .Net, gli utenti Java hanno richiesto a gran voce

l'aggiornamento del linguaggio e l'introduzione di caratteristiche funzionali avanzate. Con questi linguaggi infatti è possibile scrivere algoritmi con un numero di righe nettamente inferiore rispetto a quello che poteva fare Java. Un processo lunghissimo, iniziato con le richieste degli programmatore e successive feroci battaglie nei forum su internet, hanno portato alla definizione di cosa sarebbe dovuto rientrare nelle prossime versioni di Java. Quando la versione 6 invece di rinnovare il linguaggio puntò soprattutto nella modernizzazione di alcune librerie, in molti rimasero delusi. La delusione salì ancora di più quando Oracle comprò Sun Microsystems e dopo ben cinque anni dalla release 6, pubblicò la versione 7 estromettendo le novità più interessanti (tra cui proprio le espressioni lambda). Qualcuno aveva già definito Java un linguaggio morto, e lo slogan “Java is dead” lo si può ancora trovare oggi in molte discussioni sul web. In effetti ci eravamo abituati a release ogni due anni con Sun, cosa che con Oracle non è più accaduta sino ad ora.

Oggi però con l'avvento di Java 8 è stata apportata una vera e propria rivoluzione, e la nuova versione può essere considerata come la più innovativa tra tutte le versioni di Java. Con l'introduzione delle “espressioni lambda” e la possibilità di poter referenziare i metodi, la filosofia funzionale fa il suo ingresso nella programmazione Java! La nuova sfida è quella di far convivere i due paradigmi nelle nostre applicazioni, in modo tale da ottenere il meglio della programmazione. Non sarà facile, ed avvertiamo il lettore (anche quello già esperto) che questo modulo potrebbe risultare particolarmente ostico.

15.1 Espressioni lambda

Nel ventesimo secolo visse il famoso matematico Alonzo Church, docente di Princeton che ebbe tra l'altro studenti come Alan Turing. Church è ricordato soprattutto per la definizione del “Lambda Calcolo”, un sistema formale (ovvero una dimostrazione) sulla effettiva calcolabilità delle funzioni. La sua dimostrazione è alla base dei concetti di funzione e di ricorsività, strumenti essenziali della programmazione funzionale. Nella sua opera, Church usò l'undicesima lettera dell'alfabeto greco λ (Lambda) per evidenziare i parametri delle funzioni. Il nome “espressione Lambda” deriva quindi dal Lambda Calcolo da cui ebbe inizio in qualche modo la storia della programmazione.

Sembra che Church fu ispirato ad utilizzare la lettera λ da una delle opere fondamentali della matematica moderna: il “Principia Mathematica” di Alfred Whitehead e Bertrand Russel. In questo testo si usava il “simbolo di accento” ^, per evidenziare le variabili e questo ispirò Church ad utilizzare in primo luogo la lettera lambda maiuscola Λ , salvo poi ripiegare per qualche motivo sulla lettera minuscola λ .

Un'**espressione lambda** è detta anche **funzione anonima** (in inglese **anonymous function**). Questo perché in effetti si tratta proprio di una funzione. Non un metodo appartenente ad una classe e chiamato tramite un oggetto, ma una funzione senza nome definita al volo in maniera simile a come abbiamo fatto con le classi anonime.

Per esempio con la seguente classe anonima:

```
new Thread(new Runnable() {
```

```
@Override  
public void run() {  
    System.out.println("Prima di Java 8: Classe anonima");  
}  
}).start();
```

abbiamo fatto partire un thread passandogli come oggetto `Runnable`, un'implementazione creata al volo con la sintassi di classe anonima, ridefinendo il metodo `run()`. Siamo abituati oramai a questo tipo di sintassi che sino ad ora ha rappresentato il modo più sintetico per fare un override al volo. Il problema è che se c'è solo un metodo di cui fare override, questo tipo di sintassi sembra comunque troppo verbosa. Con Java 8 possiamo riscrivere il frammento di codice precedente usando un'espressione lambda (evidenziata in grassetto) al posto della classe anonima:

```
new Thread(()->System.out.println("Java 8: Funzione anonima")).start();
```

riducendo da cinque ad un'unica riga il nostro codice.

Le espressioni lambda sono anche note come “closures” (in italiano dovremmo tradurre “chiusure”, ma è un termine che viene usato solitamente in inglese). Precisamente una closure è un'espressione lambda che fa uso di variabili che non sono parametri e non sono variabili locali al blocco di codice che definisce l'espressione. Tecnicamente anche le classi anonime erano delle closure, ma le espressioni lambda sono closure più sintetiche e simili a quelle definite in altri linguaggi.

15.1.1 Sintassi

La sintassi di un'espressione lambda è forse la più complicata tra le sintassi di elementi definite da Java. Infatti è possibile sintetizzare questa sintassi omettendo tutto quanto il compilatore può dedurre da solo. In generale però la sintassi è la seguente:

```
([lista di parametri]) -> {blocco di codice}
```

Il primo elemento è una lista di parametri. Nell'esempio precedente la lista di parametri era vuota, ed è stata esplicitata con un blocco di parentesi tonde vuote. Poi segue una “freccia”, costituita dal segno di sottrazione “`_`”, concatenato con il simbolo maggiore “`>`”. L'ultima parte è costituita dal blocco di codice della funzione da eseguire. Si noti che nel nostro primo esempio abbiamo evitato di circondare il blocco di codice con delle parentesi graffe, visto che si tratta di un'unica istruzione. Saremmo stati obbligati ad usare le parentesi graffe nel caso il blocco di codice fosse stato costituito da più di una riga.

Ma come è possibile scrivere un'espressione lambda passandola al costruttore della classe `Thread`? Come fa il compilatore a capire il significato di quello che stiamo scrivendo? Il segreto è che il costruttore di `Thread` accetta un'istanza di `Runnable` che è un'interfaccia funzionale.

Ricordiamo ancora una volta che un'interfaccia è detta funzionale quando ha un unico metodo definito astratto (ma potrebbe contenere anche tanti metodi di default . . . implementati, metodi statici, costanti e così via). Questo metodo è a volte definito identificato come metodo SAM (che sta per “single abstract method” ovvero “singolo metodo astratto”).

L'interfaccia `Runnable` definisce il metodo `run()`, e con la sintassi dell'espressione lambda il compilatore è capace di dedurre automaticamente che stiamo riscrivendo proprio quel metodo. Infatti `run()` è l'unico metodo astratto definito in `Runnable`. In pratica il compilatore ha dedotto automaticamente che al costruttore della classe `Thread` avremmo dovuto passare un'istanza dell'interfaccia funzionale `Runnable`, e che l'espressione lambda va a sostituire l'implementazione del suo unico metodo astratto `run()`. Quindi dovremmo parlare di doppia deduzione.

Di fatto, l'unica cosa che può fare un'espressione lambda è **sostituire l'implementazione di un'interfaccia funzionale**, ed è forse questa la sua definizione più corretta.

Infatti il seguente statement è valido:

```
Runnable r = () -> System.out.println("Java 8: Funzione anonima");
```

Che di fatto ci riporta a pensare all'espressione lambda, non più come una funzione, ma come un oggetto.

15.1.2 Comprendere le espressioni lambda

Quando si ha a che fare con le espressioni lambda all'inizio si può rimanere spiazzati. La sintassi come abbiamo visto per poter essere il più compatta possibile ci permette di evitare di scrivere parentesi tonde, tipi che si possono dedurre, comandi di tipo `return` e parentesi graffe. Se da un lato scriviamo meno codice, dall'altro potremmo perdere in leggibilità. Il segreto per non perdere la bussola leggendo il codice di questo modulo (e di quelli successivi) e non avere l'impressione di leggere sintassi non Java, è quello di non dimenticare mai l'equivalenza tra classe anonima ed espressione lambda. Una volta fatto proprio il concetto che un'espressione lambda possa essere referenziata con un `reference` di un'interfaccia funzionale, e che la sua definizione non è altro che l'implementazione dell'unico metodo astratto (SAM) dell'interfaccia funzionale, siamo già ad un ottimo punto per una buona comprensione di questo argomento. Ripetiamo, l'unica cosa che può fare una'espressione lambda è sostituire un'interfaccia funzionale.

15.1.3 Espressione Lambda vs Classe Anonima

Abbiamo quindi appena asserito che il concetto fondamentale che è alla base delle espressioni lambda, è che queste hanno l'unico obiettivo di sostituire un'interfaccia funzionale. Abbiamo visto che questo però lo può fare anche una classe anonima, quindi ora cercheremo di capire le differenze tra i due approcci per trarne delle conclusioni.

15.1.3.1 Sinteticità

Il vantaggio principale nell'utilizzare un'espressione lambda in luogo di una classe anonima, risiede essenzialmente nella sinteticità dell'espressione. Nell'esempio precedente abbiamo visto come sia possibile omettere le parentesi graffe del blocco di codice quando questo è costituito da un'unica riga. Ma è anche possibile omettere il tipo dei parametri quando non c'è possibilità di errore. Inoltre è anche possibile omettere le parentesi tonde che circondano la lista dei parametri, nel caso quest'ultima fosse costituita da un unico elemento. Nel caso il blocco di codice preveda solo un'istruzione di tipo `return`, allora la parola `return` è superflua e non si deve proprio usare (pena un errore in compilazione). Quindi le espressioni lambda sono state progettate per essere il più possibile sintetiche, e permettere al programmatore di scrivere meno codice.

Per esempio consideriamo l'interfaccia funzionale `ActionListener` del package `java.awt`. Questa definisce un unico metodo astratto `actionPerformed()` che prende in input un oggetto di tipo `ActionEvent`. È possibile "registrarre" (ovvero associare) ad un certo elemento grafico (supponiamo un pulsante `javax.swing.JButton`) un'implementazione di `ActionListener`, tramite il metodo `addActionListener()`. Dopo avere effettuato quest'associazione, ogni volta che si farà click sul pulsante sarà automaticamente chiamato dalla JVM il metodo `actionPerformed()`. Prima di Java 8 dovevamo scrivere nella migliore delle ipotesi (ovvero con una classe anonima):

```
 JButton button1 = new JButton("Class Anonima");
button1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Prima di Java 8: Classe anonima ");
    }
});
```

Ora invece è possibile semplificare la sintassi in questo modo:

```
 JButton button2 = new JButton("Espressione lambda");
button2.addActionListener(
    e->System.out.println("Java 8: Funzione anonima"));
```

Si noti che abbiamo omesso sia il tipo (`ActionEvent`) del parametro sia le parentesi tonde che solitamente circondano la lista dei parametri, nonché il blocco di codice per il corpo del metodo!

Alle interfacce grafiche con le librerie Swing e AWT e la relativa gestione degli eventi è dedicata l'appendice Q on line. L'ultimo modulo di questo libro invece è dedicato alla creazione di interfacce grafiche con JavaFX.

15.1.3.2 Regole e visibilità

Un altro vantaggio che avremo utilizzando le espressioni lambda in luogo delle classi anonime, è

quello di non avere più a che fare con le complesse leggi che regolano l'utilizzo di queste ultime (cfr. paragrafo 9.1.2). Per esempio, all'interno di una classe anonima, il reference `this` si riferisce all'oggetto corrente istanziato dalla classe anonima. Per riferirsi ai membri della classe che include la classe anonima dobbiamo utilizzare una sintassi tipo:

```
NomeClasseEsterna.this.nomeMembro
```

Con le espressioni lambda invece, il reference `this` si riferisce direttamente alla classe in cui è inclusa l'espressione. Per esempio la seguente classe:

```
public class LambdaThis {  
    private String stringa = "variabile d'istanza della classe";  
    public void metodoContenenteLambda() {  
        String stringa = "variabile locale del metodo contenente";  
        new Thread(() -> System.out.println(this.stringa)).start();  
        new Thread(() -> System.out.println(stringa)).start();  
    }  
    public static void main(String args[]) {  
        LambdaThis lambdaThis = new LambdaThis();  
        lambdaThis.metodoContenenteLambda();  
    }  
}
```

una volta eseguita stamperebbe:

```
variabile d'istanza della classe  
variabile locale del metodo contenente
```

Per quanto riguarda le variabili locali e i parametri locali dichiarati all'interno di un'espressione lambda, esse condividono il loro scope (visibilità) con il blocco di codice dove sono definite. Questo significa che se riscrivessimo il metodo `metodoContenenteLambda()` del precedente esempio in questo modo:

```
public void metodoContenenteLambda() {  
    String stringa = "variabile locale del metodo contenente";  
    new Thread(() -> {  
        String stringa = "variabile locale nell'espressione lambda";  
        System.out.println(stringa);  
    }).start();  
}
```

otterremmo il seguente errore in compilazione:

```
D:\java8\Codice\modulo_15\esempi\LambdaThis.java:8: error: variable  
stringa is already defined in method metodoContenenteLambda()  
    String stringa = "variabile locale nell'espressione lambda";  
           ^
```

1 error

Infatti è come se avessimo dichiarato due volte la variabile `stringa`.

Come per le classi anonime anche le espressioni lambda possono utilizzare variabili dichiarate al di fuori dell'espressione se e solo se queste sono dichiarate `final` o sono effettivamente non modificabili (cfr. paragrafo 9.1.2). Quindi il seguente frammento di codice:

```
public void startCount() {  
    int count = 0;  
    new Thread(() -> {  
        while (count < 100) {  
            System.out.println(count++);  
        }  
    }).start();  
}
```

produrrà il seguente output in fase di compilazione:

```
D:\java8\Codice\modulo_15\esempi\TestCountArrayTrick.java:5: error:  
local variables referenced from a lambda expression must be final or  
effectively final  
        while (count < 100) {  
                ^
```

Questo avviene in quanto abbiamo provato a modificare una variabile del metodo `startCount()` con l'operatore di post-incremento.

In molti casi il compilatore si accorgerà se proviamo a modificare una variabile dichiarata esternamente all'espressione lambda, ma non possiamo farvi affidamento completamente. Per esempio se la variabile in questione è di tipo `reference`, allora anche dichiarandola `final`, potremmo sempre modificarne la struttura interna senza che il compilatore se ne accorga. Infatti un famoso trucco per far funzionare l'esempio precedente, è “wrappare” il valore di `count` in un array (che è un oggetto) di dimensione 1 come nel seguente frammento di codice:

```
public void startCount() {  
    int count = 0;  
    int array[] = new int[1];  
    array[0] = count;  
    new Thread(() -> {  
        while (array[0] < 100) {  
            System.out.println(array[0]);  
            array[0]++;  
        }  
    }).start();  
}
```

Inoltre ogni variabile d'istanza è utilizzabile e modificabile. Per questo bisogna stare attenti a non utilizzare le espressioni lambda in maniera troppo superficiale. Per esempio il seguente codice:

```

public class TestCount {
    private int count;
    public TestCount (int count) {
        this.count = count;
    }
    public void startCount() {
        new Thread(() -> {
            while (count < 100) {
                System.out.println(count++);
            }
        });
    }
}

```

non è thread-safe perché il metodo `startCount()` potrebbe essere chiamato sullo stesso oggetto e provocare problemi di concorrenza.

15.1.3.3 Dinamicità

Nel paragrafo 9.2.1 abbiamo mostrato come utilizzare le classi anonime per rendere più dinamico il nostro codice ed evitare copia incolla. In particolare abbiamo mostrato tramite un procedimento come poter “passare del codice” da eseguire ad un metodo che lo eseguirà. Ma le espressioni lambda sono l’ideale per poter passare codice! Riprendiamo quindi l’esempio che abbiamo descritto alla fine del paragrafo 9.2.1, dove avevamo la necessità di filtrare i titoli dei film di una videoteca secondo dei criteri di selezione (nel nostro caso film con voto di recensione più alto di 3, e film di fantascienza). Ricordiamo che avevamo creato la classe `Film`:

```

public class Film {
    private String nome;
    private String genere;
    private int mediaRecensioni;

    public Film (String nome, String genere, int mediaRecensioni) {
        this.nome = nome;
        this.genere = genere;
        this.mediaRecensioni = mediaRecensioni;
    }
    // Metodi set e get omessi . . .
    public String toString(){
        return getNome();
    }
}

```

l’interfaccia (che ora possiamo anche annotare come interfaccia funzionale) `FiltroFilm`:

```

@FunctionalInterface
public interface FiltroFilm {

```

```
        boolean filtra(Film film);  
    }  
}
```

e la classe Videoteca:

```
public class Videoteca {  
    private Film[] films;  
  
    public Videoteca () {  
        films = new Film[10];  
        caricaFilms();  
    }  
    public void setFilms(Film[] films) {  
        this.films = films;  
    }  
  
    public Film[] getFilms() {  
        return films;  
    }  
  
    public Film[] getFilmFiltrati(FiltroFilm filtroFilm) {  
        Film [] filmFiltrati = new Film[10];  
        for (int i = 0, j= 0; i< 10;i++) {  
            if (filtroFilm.filtra(films[i])) {  
                filmFiltrati[j] = films[i];  
                j++;  
            }  
        }  
        return filmFiltrati;  
    }  
    private void caricaFilms() {  
        //Caricamento film...  
    }  
}
```

Infine avevamo richiamato il metodo `getFilmFiltrati()` passando due diverse implementazioni di `FilmFiltro` usando la notazione di classe anonima nel seguente modo:

```
Videoteca videoteca = new Videoteca();  
System.out.println("Bei Film:");  
Film[] beiFilms = videoteca.getFilmFiltrati(new FiltroFilm() {  
    @Override  
    public boolean filtra(Film film) {  
        return film.getMediaRecensioni() > 3;  
    }  
});  
System.out.println("\nFilm di Fantascienza:");  
Film[] filmDiFantascienza = videoteca.getFilmFiltrati(new FiltroFilm() {  
    @Override
```

```
        public boolean filtra(Film film) {
            return "Fantascienza".equals(film.getGenere());
        }
    });
}
```

Ora che conosciamo la sintassi delle espressioni lambda possiamo riscrivere il precedente frammento di codice in maniera più compatta:

```
Videoteca videoteca = new Videoteca();
System.out.println("Bei Film:");
Film[] beiFilms = videoteca
    .getFilmFiltrati((Film film)-> film.getMediaRecensioni() > 3);
System.out.println("\nFilm di Fantascienza:");
Film[] filmDiFantascienza = videoteca
    .getFilmFiltrati((Film film)-> "Fantascienza".equals(film.getGenere()));
```

Si noti che in un metodo normale avremmo usato anche un comando `return` come abbiamo fatto quando abbiamo creato la classe anonima precedente. Invece la natura sintetica della sintassi delle espressioni lambda richiede che il comando `return` sia omesso.

Volendo possiamo anche creare altri filtri al volo in maniera molto semplice. Per esempio di seguito filtriemo i film che hanno un nome che termina con la lettera “s”:

```
Film[] filmCheFinisconoConS = videoteca
    .getFilmFiltrati((Film film)->film.getName().endsWith("s"));
```

Anche se tutto questo si poteva già fare con le classi anonime, sembra evidente che le espressioni lambda siano decisamente più adeguate a risolvere questo tipo di problemi.

15.1.3.4 Gestione delle eccezioni

Quando utilizziamo le espressioni lambda dobbiamo fare attenzione alle cosiddette checked exception, ovvero alle eccezioni che non sono sottoclassi di `RuntimeException`. Per tale motivo dovrebbero essere dichiarate nella clausola `throws` della dichiarazione del metodo che potrebbe lanciare l’eccezione nel caso non siano gestite (cfr. modulo 8). Ma le espressioni lambda non dichiarano metodi, e non ci sarebbe spazio per utilizzare una clausola `throws` se non nel metodo dell’interfaccia funzionale che rappresentano. Ma questo non è sempre possibile. Cerchiamo di spiegarci con un semplice esempio. Il seguente frammento di codice:

```
new Thread(()-> {
    Thread.sleep(1000);
    System.out.println("Hello World");
}).start();
```

darà luogo ad un errore in compilazione:

```
D:\java8\Codice\modulo_15\esempi\LambdaException.java:4: error:  
unreported exception InterruptedException; must be caught or declared  
to be thrown  
        Thread.sleep(1000);  
        ^  
1 error
```

Abbiamo due possibilità per risolvere il problema, o gestire l'eccezione con un `try-catch`:

```
new Thread(() -> {  
    try {  
        Thread.sleep(1000);  
    }  
    catch (InterruptedException exc) {  
        exc.printStackTrace();  
    }  
    System.out.println("Hello World");  
} ).start();
```

oppure utilizzare un'interfaccia funzionale che definisca il metodo astratto con la clausola `throws` richiesta. Nel nostro caso non possiamo farlo con l'interfaccia `Runnable`, perché la definizione del metodo `run()` non dichiara una clausola `throws`, e per le proprietà dell'override, non possiamo definire una sottointerfaccia che aggiunge la clausola `throws` (cfr. paragrafo 8.5.2). Dobbiamo ripiegare quindi su un meccanismo un po' più complesso: usare l'interfaccia (funzionale) `Callable` e il suo metodo `call()` che invece dichiara una clausola `throws` ad `Exception`, e passarlo in input ad un metodo `submit()` di un `ExecutorService`, invece di lanciare il comando `start()` di un `Thread` come nell'esempio precedente. Potremmo per esempio creare la seguente interfaccia funzionale:

```
@FunctionalInterface  
interface MyCallable extends Callable<Void> {  
    @Override  
    Void call() throws InterruptedException;  
}
```

Con la quale abbiamo esteso l'interfaccia generica `Callable` passandogli il tipo `Void`. Questo implicherà che il metodo `call()` ritornerà un tipo `Void`.

Un oggetto `void` è come se fosse una classe wrapper per il valore di ritorno `void` dei metodi.

Quindi abbiamo riscritto il metodo `call()` in modo tale che dichiari nella sua clausola `throws` la `InterruptedException` che l'espressione lambda dovrà gestire. Adesso possiamo riscrivere il frammento di codice contenente l'espressione lambda in questo modo:

```
MyCallable callable = () -> {
```

```
        Thread.sleep(1000);
        System.out.println("Hello World");
        return null;
    } ;
ExecutorService pool = Executors.newFixedThreadPool(1);
pool.submit(callable);
pool.shutdown();
```

In particolare abbiamo assegnato l'espressione lambda ad un'istanza della nostra estensione `MyCallable`. Si noti che siamo stati costretti a ritornare `null`, perché comunque il metodo si aspetta un tipo di ritorno `Void`. Questo righe quindi ora compileranno perché il metodo `call()` che stiamo definendo con l'espressione lambda definisce la clausola `throws` richiesta. Poi istanziamo un `ExecutorService pool` di dimensione 1, e invochiamo `submit()` passandogli l'espressione lambda appena definita. Infine chiudiamo l'executor `pool`.

Sicuramente queste situazioni non si adattano molto bene alla natura sintattica delle espressioni lambda. Si noti che ci saremmo pure potuti risparmiare la definizione dell'interfaccia `MyCallable`, ed avremmo potuto scrivere direttamente:

```
Callable<Void> callable = ()-> {
    Thread.sleep(1000);
    System.out.println("Hello World");
    return null;
};
```

visto che il metodo `call()` di `Callable` ha una clausola `throws` generica (`Exception`), ma in questo modo altre eccezioni checked non sarebbero state più sollevate dal compilatore nel caso ce ne fosse stato bisogno.

15.1.3.5 Quando usare le espressioni lambda

Dovremmo usare le espressioni lambda quando il nostro obiettivo è quello di passare in maniera dinamica un certo algoritmo ad un altro metodo. Questo ci serve per eseguire l'algoritmo in un contesto definito dal metodo a cui stiamo passando l'algoritmo. Passiamo del codice ad un metodo, per dare a quest'ultimo la responsabilità di chiamarlo al momento giusto. Un buon esempio è quando abbiamo passato un'espressione lambda al metodo `addActionListener()`, con lo scopo di eseguirla alla pressione di un pulsante su un'interfaccia grafica. Potremmo anche passare del codice ad un metodo per farlo eseguire in base ad un altro evento come il lancio di un'eccezione. Oppure potremmo anche fare in modo che il nostro codice sia eseguito mentre si itera su ogni elemento di un array. In generale passare un'espressione lambda, significherà far decidere al metodo a cui abbiamo passato il codice, il momento in cui eseguirlo.

Inoltre dobbiamo pensare sempre al fatto che usare un'espressione lambda significa sostituire un'interfaccia funzionale in maniera sintetica.

Nel caso il nostro obiettivo invece sia quello di estendere le funzionalità o aggiungere dati all'astrazione offerta di un'interfaccia funzionale, allora sarebbe preferibile utilizzare una classe

anonima.

15.2 Reference a metodi

Quindi con Java 8 abbiamo la possibilità di chiamare funzioni anonime. Le dovremmo preferire sicuramente alle classi anonime nel caso il loro blocco di codice contenga pochi statement. Molto spesso però, esse contengono semplicemente un'unica chiamata ad un metodo. In casi come questo, esiste una sintassi ancora più compatta delle espressioni lambda. Infatti con Java 8 possiamo utilizzare reference a metodi esistenti. In questo caso parliamo di **reference a metodi** (in inglese **method references**).

Riprendiamo l'esempio della videoteca di prima. Se creassimo questa classe che definisce dei filtri:

```
public class Filtri {  
    public static boolean isBelFilm(Film film) {  
        return film.getMediaRecensioni() > 3;  
    }  
  
    public static boolean isFilmDiFantascienza(Film film) {  
        return "Fantascienza".equals(film.getGenere());  
    }  
}
```

potremmo passare l'implementazione dei metodi usando la sintassi dei reference ai metodi nel seguente modo:

```
Film[] filmDiFantascienza = videoteca  
    .getFilmFiltrati(Filtri::isFilmDiFantascienza);  
stampaFilm(filmDiFantascienza);
```

L'effetto è lo stesso delle espressioni lambda, ma la sintassi è ancora più semplice e compatta. Infatti in questo caso abbiamo specificato solo il nome della classe (`Filtri`) il simbolo `::` e il nome del metodo (`isFilmDiFantascienza`) senza neanche specificare le parentesi (ed eventualmente i parametri). Attenzione che non stiamo semplicemente chiamando il metodo `isFilmDiFantascienza()`, altrimenti avremmo direttamente scritto:

```
Film[] filmDiFantascienza = videoteca  
    .getFilmFiltrati(Filtri.isFilmDiFantascienza());
```

ma questo codice non avrebbe compilato perché `isFilmDiFantascienza()` ritorna un booleano e non possiamo passarlo al metodo `getFilmFiltrati()`. Infatti quest'ultimo si aspetta in input un oggetto di tipo `FiltroFilm`. Insomma, **con le espressioni lambda sostituivamo un'istanza di un'interfaccia funzionale passandogli l'implementazione dell'unico metodo definito. Con la sintassi del reference a metodo andiamo a sostituire un'istanza di un'interfaccia funzionale, passandogli il nome di un metodo già definito, che corrisponde per tipo di ritorno e parametri in**

input all’unico metodo SAM definito nell’interfaccia funzionale. Anche in questo caso la deduzione automatica dei tipi fa tutto il lavoro per noi.

Inoltre questo tipo di sintassi si fa preferire rispetto ad un’espressione lambda nel caso abbiam bisogno di utilizzare più volte la chiamata al reference method. Infatti con un’espressione lambda spesso il vantaggio è quello di creare del codice al volo. Ma se questo codice dobbiamo usarlo più volte in classi diverse, probabilmente inizieremmo a far uso dei copia incolla. E come abbiamo già asserito in precedenza il copia-incolla è una pratica estremamente sconveniente.

15.2.1 Sintassi

La sintassi dei reference a metodi può essere di quattro tipi:

1. **Reference a un metodo statico** (in inglese **reference to a static method**):

```
NomeClasse::nomeDelMetodoStatico
```

2. **Reference a un metodo d’istanza** (in inglese **reference to an instance method of a particular object**):

```
NomeOggetto::nomeDelMetodoDIstanza
```

3. **Reference a un metodo d’istanza di un certo tipo** (in inglese **reference to an instance method of an arbitrary object of a particular type**):

```
NomeTipo::nomeDelMetodoDIstanza
```

4. **Reference a un costruttore** (in inglese **reference to a constructor**):

```
NomeClasse::new
```

Cercheremo di capire in dettaglio le differenze tra queste quattro tipologie di reference a metodo nei prossimi paragrafi.

15.2.2 Reference a un metodo statico

Abbiamo appena visto un esempio di reference a metodo statico con la classe `Filtri`. In particolare abbiamo già visto come questo tipo di reference a metodo sia del tutto equivalente ad una espressione lambda, tanto che l’abbiamo usato proprio allo stesso modo. La differenza che abbiamo già fatto notare, risiede nel fatto che un reference ad un metodo, usa un metodo già creato e non si crea al volo la sua implementazione come con le espressioni lambda. Nel caso esistesse la classe `Filtri` avremmo comunque potuto usare una espressione lambda con la seguente sintassi:

```
Film[] filmDiFantascienza = videoteca  
    .getFilmFiltrati((Film film) -> Filtri.isBelFilm(film));
```

ma è più semplice e compatto usare un reference a metodo come abbiamo visto prima:

```
Film[] filmDiFantascienza = videoteca
    .getFilmFiltrati(Filtri::isFilmDiFantascienza);
```

15.2.3 Reference a un metodo d'istanza

Lo stesso discorso fatto per un reference a metodo statico può essere ripetuto per i reference a metodi d'istanza. L'unica differenza è che il metodo coinvolto non è statico, quindi come reference bisogna utilizzare un oggetto istanziato dalla classe che definisce il metodo. Per esempio, creiamo una classe che fornisca dei metodi per ordinare i film:

```
public class OrdinamentoFilm {
    public int ordinaPerNome(Film film1, Film film2) {
        return film1.getNome().compareTo(film2.getNome());
    }

    public int ordinaPerMediaRecensioni(Film film1, Film film2) {
        Integer mediaRecensioniFilm1 =
            new Integer(film1.getMediaRecensioni());
        Integer mediaRecensioniFilm2 =
            new Integer(film2.getMediaRecensioni());
        return mediaRecensioniFilm2.compareTo(mediaRecensioniFilm1);
    }
}
```

Per gestire l'ordinamento abbiamo creato dei metodi che sfruttano il metodo `compareTo()` di `String` (ereditati dall'interfaccia funzionale `java.lang.Comparable` che abbiamo approfondito nel paragrafo 11.1.9). Questa classe ci offre due tipi di ordinamento: alfabetico per il nome, e per media recensioni dal voto più alto al più basso. Possiamo quindi ordinare i film nel seguente modo:

```
Film[] films = videoteca.getFilms();
System.out.println("\nFilm ordinati per nome con un reference "
    + "a metodo d'istanza:");
OrdinamentoFilm ordinamentoFilm = new OrdinamentoFilm();
Arrays.sort(films, ordinamentoFilm::ordinaPerNome);
stampaFilm(films);
System.out.println("\nFilm ordinati per media recensioni con "
    + "un reference a metodo d'istanza:");
Arrays.sort(films, ordinamentoFilm::ordinaPerMediaRecensioni);
stampaFilm(films);
```

È possibile fare questo perché indipendentemente dal nome del metodo che stiamo passando, il metodo `sort()` vuole in input il blocco di codice che deve rappresentare l'implementazione del metodo `compare()` della classe `Comparator`. Il compilatore trovando corrispondenza tra parametri in input e output consente questa sintassi grazie alla deduzione automatica. Anche questa volta quindi potevamo usare un'espressione lambda invece di un reference a metodo nel seguente modo:

```
Arrays.sort(films,
    (Film film1, Film film2) -> film1.getNome().compareTo(film2.getNome()));
```

ma il reference method è risultato ancora una volta più compatto.

È anche possibile usare i reference `this` e `super` usando questa sintassi. Infatti è possibile usare questi reference anche nelle espressioni lambda equivalenti. Per esempio se avessimo la classe:

```
public class Saluto {
    public void saluta() {
        System.out.println("Ciao");
    }
}
```

potremmo estenderla con la seguente ed utilizzare un reference al metodo sfruttando `super`:

```
public class SalutoAsincrono extends Saluto {
    @Override
    public void saluta() {
        new Thread(super::saluta).start();
    }
}
```

15.2.4 Reference a un metodo d'istanza di un certo tipo

Questo tipologia di sintassi all'inizio può portare facilmente a confondersi, perché dopo aver visto i due precedenti tipi di reference a metodo, questo può sembrare un po' un mix tra i due. Infatti sembra di chiamare un metodo non statico con il nome di una classe, ma non è così che bisogna intendere la sintassi:

```
NomeTipo::nomeDelMetodoDIstanza
```

Se abbiamo un metodo d'istanza da cui il compilatore può dedurre parametri di input e di output, è possibile usare il nome del tipo invece del nome di un oggetto. Per esempio con le seguenti istruzioni:

```
List<String> filmNames = new ArrayList<String>();
for(Film film:films) {
    filmNames.add(film.getNome());
}
Collections.sort(filmNames, String::compareToIgnoreCase);
stampaFilm(films);
```

abbiamo creato un lista `filmNames` di stringhe contenente i nomi dei film dell'esempio precedente. Poi abbiamo ordinato la collezione passando il reference al metodo `compareToIgnoreCase()` della classe `String`, che ordina le stringhe ignorando se sono composte da caratteri maiuscoli o minuscoli. Abbiamo usato però una sintassi del tipo:

```
NomeDellaClasse::nomeDelMetodoDIstanza
```

Si noti che il metodo `compareToIgnoreCase()` non è statico, quindi è una sintassi diversa dal primo caso che abbiamo visto. Infatti il metodo d'istanza `compareToIgnoreCase()` è definito così:

```
public int compareToIgnoreCase(String str) {/. . .}
```

Quindi se dovessimo usare questo metodo normalmente con delle stringhe potremmo usarlo in questo modo:

```
String s1 = "Rosalia";
String s2 = "rosalia";
System.out.println("\nRosalia vs rosalia = "
+ s1.compareToIgnoreCase(s2));
```

Se dovessimo scrivere quindi l'equivalente istruzione con una espressione lambda, dovremmo scrivere:

```
Collections.sort(filmNames, (s1,s2)->s1.compareToIgnoreCase(s2));
```

Con l'espressione lambda abbiamo potuto evitare di scrivere i tipi di `s1` e `s2` (che potevamo chiamare anche `x` ed `y`, sono solo degli identificatori) perché il compilatore dal contesto ha potuto dedurre che si trattava di stringhe. Con la sintassi del reference a metodo che abbiamo usato nell'esempio, specificando `String::compareToIgnoreCase`, abbiamo potuto evitare di specificare l'implementazione del metodo, e di conseguenza specificare `s1` e `s2` non avrebbe avuto senso perché il compilatore può dedurlo automaticamente. Però abbiamo dovuto specificare `String`, per indicare al compilatore che ci stavamo riferendo al metodo di `String`, e non ad un altro metodo in un'altra classe. A quel punto il compilatore ha potuto dedurre che nel metodo `sort()`, potrà usare gli elementi della collezione `filmNames` con l'implementazione di `compareToIgnoreCase()` definito nella classe `String`. In definitiva è stato necessario specificare solo il tipo da cui prendere il metodo specificato, non occorre specificare oggetti di tipo stringa perché non dobbiamo definire al volo un'implementazione del metodo già esistente.

15.2.5 Reference a un costruttore

È possibile anche referenziare un costruttore. È molto semplice, è sufficiente usare la parola chiave `new` come nome del metodo da chiamare. In effetti un costruttore è come un metodo che ritorna un oggetto della sua classe, ecco perché ha senso assegnargli un reference. Passiamo come al solito ad un esempio. Useremo una factory, ovvero una classe che ha il compito di creare oggetti (definizione semplificata di uso comune del design pattern “Factory Method”). Supponiamo di avere una semplice classe `Chitarra` come la seguente:

```
public class Chitarra {
    private String marca;
```

```
    public Chitarra (String marca) {
        this.marca = marca;
        System.out.println("Creata chitarra: " + marca);
    }
}
```

Come interfaccia funzionale consideriamo la seguente:

```
@FunctionalInterface
public interface FabbricaChitarra {
    Chitarra getChitarra(String marca);
}
```

In una situazione pre-Java 8 dovremmo creare un'implementazione dell'interfaccia:

```
public class FabbricaChitarraImpl implements FabbricaChitarra {
    @Override
    public Chitarra getChitarra(String marca) {
        return new Chitarra(marca);
    }
}
```

per poi utilizzarla con un codice simile al seguente:

```
FabbricaChitarra fabbricaChitarra = new FabbricaChitarraImpl();
Chitarra chitarra1 = fabbricaChitarra.getChitarra("Fender");
```

Con Java 8 possiamo completamente evitare di creare la classe factory `FabbricaChitarraImpl` usando un reference a costruttore:

```
FabbricaChitarra fabbricaChitarra = Chitarra::new;
Chitarra chitarra1 = fabbricaChitarra.getChitarra("Fender");
```

Anche in questo caso è possibile utilizzare un'espressione lambda in luogo del reference a costruttore nel seguente modo:

```
FabbricaChitarra fabbricaChitarra =
    (marca) -> { return new Chitarra(marca); };
Chitarra chitarra1 = fabbricaChitarra.getChitarra("Fender");
```

ma anche in questo caso è preferibile la sintassi del reference a costruttore.

15.3 Le interfacce funzionali del package `java.util.function`

Abbiamo visto come le espressioni lambda e i reference a metodi si utilizzino per implementare comportamenti definiti in interfacce funzionali. A parte quando abbiamo fatto esempi con `ActionListener` e `Runnable`, sino ad ora abbiamo creato noi le interfacce funzionali che ci

servivano. In realtà esistono anche altre interfacce funzionali che possiamo usare, come `Iterable` con il suo metodo `iterator()` ed `Executor` con il suo metodo `execute()`. Java 8 ha però introdotto il package `java.util.function`, che definisce una serie di interfacce funzionali che dovrebbero coprire quasi sempre i nostri bisogni. Quindi in realtà creeremo solo raramente delle interfacce funzionali personali. Di seguito illustriamo le interfacce funzionali più importanti, esse sono dichiarate tutte generiche. Il lettore potrà trovare altre decine di interfacce funzionali all'interno di questo package.

15.3.1 Predicate

Quest'interfaccia funzionale è dichiarata nel seguente modo (senza metodi di default, metodi statici, import, package e commenti):

```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T t);  
}
```

Come si può intuire dall'unico metodo astratto, è un'interfaccia perfetta per fare dei test che ritornano un booleano. Il tipo parametrazione `T` ci garantisce la massima flessibilità sul parametro in input che può essere deciso al volo. Se riprendiamo ancora l'esempio della videoteca, potremmo eliminare del tutto l'interfaccia `FiltroFilm`, e usare al suo posto `Predicate`. Quindi dovremmo riscrivere solo il metodo `getFilmFiltrati()` che avevamo definito nella classe `Videoteca` nel seguente modo:

```
public Film[] getFilmFiltrati(Predicate<Film> filtroFilm) {  
    Film [] filmFiltrati = new Film[10];  
    for (int i = 0, j = 0; i < 10; i++) {  
        if (filtroFilm.test(films[i])) {  
            filmFiltrati[j] = films[i];  
            j++;  
        }  
    }  
    return filmFiltrati;  
}
```

Si noti che abbiamo solo dovuto usare come tipo parametrazione per `Predicate` la classe `Film`, e sostituire la chiamata al metodo `getFilmFiltrati()` con il metodo `test()`. Per il resto non cambia niente.

`Predicate` è l'interfaccia funzionale ideale per filtrare oggetti, anche perché definisce ulteriori metodi per concatenare filtri: `or()`, `and()` e `xor()`.

Quindi se volessimo filtrare film di fantascienza che hanno anche un buona media di recensioni, basterebbe scrivere:

```
System.out.println("\nBei film di Fantascienza");  
Predicate<Film> predicateFilmFantascienza =  
    (Film film) ->"Fantascienza".equals(film.getGenere());
```

```

Predicate<Film> predicateBeiFilm =
    (Film film) -> film.getMediaRecensioni() > 3;
Film[] filmDiFantascienzaBelli = videoteca
    .getFilmFiltrati(predicateFilmFantascienza.and(predicateBeiFilm));
stampaFilm(filmDiFantascienzaBelli);

```

Java 8 fornisce finalmente una grande flessibilità!

15.3.2 Consumer

Quest'interfaccia funzionale è dichiarata nel seguente modo (senza metodi di default, metodi statici, import, package e commenti):

```

@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}

```

Essa è intesa per fornire funzionalità di aggiornamento dello stato interno dell'oggetto che gli viene passato, o comunque di usarlo (per esempio stamparne il metodo `toString()`). Come si può notare infatti il metodo `accept()` non ha tipo di ritorno.

Si rimarca ancora il fatto che questa è un'interfaccia generica e quindi flessibile ad accettare qualsiasi tipo di oggetto; questa è una caratteristica di tutte le interfacce funzionali del package `java.util.function`. Supponiamo di voler creare una classe che ci permetta di aggiornare al momento opportuno lo stato interno degli oggetti di tipo `Film`. Potremmo creare la seguente classe:

```

import java.util.function.Consumer;
public class SocialBacheca {
    public void aggiornaFilm(Film film, Consumer<Film> consumer) {
        consumer.accept(film);
    }
}

```

Quindi se volessimo aggiornare la media voto delle recensioni di un film potremmo scrivere semplicemente:

```

Videoteca videoteca = new Videoteca();
SocialBacheca bachecca = new SocialBacheca();
Film films[] = videoteca.getFilms();
Film lanternaVerde = films[7];
bachecca.aggiornaFilm(lanternaVerde, film -> film.setMediaRecensioni(5));
System.out.printf("La media voto delle recensioni di %s è stata "
    + "aggiornata a %s", lanternaVerde.getNome(),
    lanternaVerde.getMediaRecensioni());

```

Si noti che nella espressione lambda abbiamo usato il reference `film` (ma l'avremmo anche potuto

chiamare `x`) che il compilatore riesce a dedurre essere di tipo `Film`.

Anche nel caso di `Consumer` possiamo sfruttare il suo metodo di default `andThen()` per concatenare aggiornamenti. Per esempio con il seguente frammento di codice:

```
Film starWarsEpisodio1 = films[1];
Consumer<Film> aggiornaVotoConsumer = film -> film.setMediaRecensioni(3);
Consumer<Film> aggiornaNomeConsumer =
    film -> film.setNome("Star Wars Episodio 1 - La minaccia fantasma");
bacheca.aggiornaFilm(starWarsEpisodio1,
    aggiornaNomeConsumer.andThen(aggiornaVotoConsumer));
System.out.printf(
    "\nLa media voto delle recensioni di %s è stata aggiornata a %s",
    starWarsEpisodio1.getNome(), starWarsEpisodio1.getMediaRecensioni()
);
```

Aggioreremo sia il nome sia la media voto di un film. Ma l'aggiornamento del voto avverrà solo se andrà a buon fine l'aggiornamento del nome.

15.3.3 Supplier

Quest'interfaccia funzionale è dichiarata nel seguente modo (senza import, package e commenti):

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Essa si adatta bene nei contesti in cui serve gestire una factory. Il suo metodo `get()` restituisce un'istanza del tipo parametro usato, e quindi è facile pensare alla sua implementazione per creare e restituire un oggetto. Riprendiamo quindi l'esempio della classe `Chitarra` usato per mostrare i reference a costruttori. Aggiungiamo il metodo `suona()` alla classe `Chitarra`:

```
public class Chitarra {
    private String marca;

    public Chitarra (String marca) {
        this.marca = marca;
        System.out.println("Creata chitarra: " + marca);
    }

    public void suona() {
        System.out.printf("Sta suonando una %s ...", marca);
    }
}
```

Quindi creiamo la classe `Chitarrista` nel seguente modo:

```
import java.util.function.Supplier;
```

```
public class Chitarrista {  
    public void suonaChitarra(Supplier<Chitarra> marcaSupplier) {  
        Chitarra chitarra = marcaSupplier.get();  
        chitarra.suona();  
    }  
}
```

Si noti come il metodo `get()` restituisca un'istanza del tipo parametro `Chitarra`. Infine con il seguente frammento di codice, potremo far suonare al chitarrista la chitarra:

```
Chitarrista mrTambourine = new Chitarrista();  
Supplier<Chitarra> chitarraSupplier = () -> new Chitarra("Ibanez");  
mrTambourine.suonaChitarra(chitarraSupplier);
```

15.3.4 Function

Questa interfaccia funzionale è dichiarata nel seguente modo (senza metodi di default, metodi statici, import, package e commenti):

```
@FunctionalInterface  
public interface Function<T, R> {  
    R apply(T t);  
}
```

Quest'interfaccia ci permette di astrarre proprio il concetto di funzione più classico, dove c'è un input (definito dal tipo parametro `T`) e un output (tipo parametro `R`). Esiste anche l'interfaccia funzionale `BiFunction`, che è definita come segue (senza metodi di default, import, package e commenti):

```
@FunctionalInterface  
public interface BiFunction<T, U, R> {  
    R apply(T t, U u);  
}
```

dove gli input sono due (`T` ed `U`).

Esistono le equivalenti “Bi” interfacce anche per le altre interfacce funzionali (`BiPredicate`, `BiSupplier` e così via).

Per l'esempio useremo `BiFunction`. Supponiamo di voler creare un metodo che esegue una funzione di trasformazione di due stringhe in un'unica stringa. Essa deve quindi prendere in input due stringhe e ritornarne una. Potremmo creare il seguente metodo:

```
public static String trasformaStringhe(String s1, String s2,  
    BiFunction<String, String, String> biFunction) {
```

```
        return biFunction.apply(s1,s2);  
    }  
}
```

Se volessimo sfruttare il metodo precedente per eseguire una concatenazione di due stringhe separandole con un trattino, potremmo scrivere:

```
BiFunction<String, String, String> concatenazioneConTrattino =  
    (String s1, String s2) -> { return s1 + "-" + s2; };  
System.out.println(concatenaConTrattino("Andrea", "Simone", concatenazioneConTra
```

Si noti che nella libreria non esistono altre interfacce come `TriPredicate`, `TriFunction` e così via, ma all'occorrenza possiamo semplicemente crearle. Ecco come definire una `TriFunction`:

```
@FunctionalInterface  
public interface TriFunction<T, U, V, R> {  
    R apply(T t, U u, V v);  
}
```

15.3.5 UnaryOperator e composizione di espressioni lambda

Quest'interfaccia funzionale è dichiarata nel seguente modo (senza commenti e metodi statici):

```
@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T> {  
}
```

Come è possibile notare, `UnaryOperator` è semplicemente un'estensione dell'interfaccia funzionale `Function` e quindi eredita da essa il metodo astratto:

```
T apply(T t);
```

`UnaryOperator` astrae un'operazione che viene fatta su un singolo parametro e che produce un risultato dello stesso tipo del parametro. Essa quindi si adatta bene ad operazioni di tipo *trasformazione*. Quest'interfaccia è utile quando si ha intenzione di combinare più espressioni lambda per eseguirle in un solo colpo, evitando trasformazioni intermedie. Per esempio consideriamo la classe `Film` che abbiamo definito nel paragrafo 15.1.3.3. Creiamo una classe che astragga il concetto di programmazione del film:

```
public class ProgrammazioneFilm {  
    private Film film;  
    private LocalDate data;  
    private LocalTime orario;  
  
    public ProgrammazioneFilm (Film film, LocalDate data, LocalTime orario)  
    {  
        this.film = film;  
        this.data = data;
```

```

        this.orario = orario;
    }
    //Metodi setter e getter omessi
    public String toString() {
        return film.getNome() + " verrà proiettato il " + getData() + " alle
               " + getOrario();
    }
}

```

Una classe Cinema avrà la responsabilità della schedulazione dei film:

```

public class Cinema {
    public static UnaryOperator<ProgrammazioneFilm> componiCambi(
        UnaryOperator<ProgrammazioneFilm> op1,
        UnaryOperator<ProgrammazioneFilm> op2) {
        return x->op2.apply(op1.apply(x));
    }
    public static ProgrammazioneFilm cambiaProgrammazione(
        ProgrammazioneFilm pf,
        UnaryOperator<ProgrammazioneFilm> op) {
        return op.apply(pf);
    }
}

```

Abbiamo definito al suo interno il metodo `componiCambi()`, che compone un'operazione del primo parametro `UnaryOperator op1` con il secondo parametro `op2` per restituire sempre un oggetto `UnaryOperator`. Quindi questo metodo restituisce la composizione dei due `UnaryOperator` presi in input sotto forma sempre di uno `UnaryOperator`. Lo scopo è poi di riutilizzare l'oggetto composito per passarlo al metodo `cambiaProgrammazione()`.

È il primo esempio in cui vediamo come un'espressione lambda viene ritornata come output di un metodo, ma questo non deve meravigliarci. Le espressioni lambda possono avere un reference, e quindi alla pari dei reference possono essere sia input che output di metodi.

Quest'ultimo infatti prende in input un oggetto `ProgrammazioneFilm pf` e lo `UnaryOperator op`, e chiama il metodo `apply()` su `op`, passandogli `pf`. Il metodo ritornerà l'oggetto `pf` modificato con l'implementazione di `op`. Per comprendere meglio l'esempio, completiamolo con l'implementazione di un frammento di codice estratto da un ipotetico metodo `main()`:

```

UnaryOperator<ProgrammazioneFilm> ritardaDiUnOra= p-> {
    LocalTime orario = p.getOrario();
    orario = orario.plus(1, ChronoUnit.HOURS);
    p.setOrario(orario);
    return p;
};

```

```

UnaryOperator<ProgrammazioneFilm> ritardaDiUnGiorno= p-> {
    p.setData(p.getData().plus(1, ChronoUnit.DAYS));
    return p;
} ;
Film shining = new Film("Shining", "Thriller", 5);
ProgrammazioneFilm programmazioneShining =
    new ProgrammazioneFilm(shining, LocalDate.of(2014, 5, 29),
        LocalTime.of(21, 0));
UnaryOperator<ProgrammazioneFilm> ritardaDiUnOraEUnGiorno =
    Cinema.componiCambi(ritardaDiUnOra, ritardaDiUnGiorno);
System.out.println(Cinema.cambiaProgrammazione(
    programmazioneShining, ritardaDiUnOraEUnGiorno));

```

Abbiamo definito due `UnaryOperator` implementandoli con due semplici espressioni lambda. La prima (che definisce `ritardaDiUnOra`) incrementa l'orario di un oggetto `ProgrammazioneFilm` di un'ora. La seconda (che definisce `ritardaDiUnGiorno`) incrementa la data di un oggetto `ProgrammazioneFilm` di un giorno, ma abbiamo implementato il suo codice in maniera più compatta (e meno leggibile). Poi abbiamo istanziato un oggetto di tipo `Film` che abbiamo chiamato `shining`. Subito dopo abbiamo creato l'oggetto `programmazioneShining` di tipo `ProgrammazioneFilm` che definisce per `shining` un orario e una data. Poi abbiamo ottenuto un oggetto `UnaryOperator` composto dalle due espressioni lambda che abbiamo creato precedentemente, e l'abbiamo chiamato `ritardaDiUnOraEUnGiorno`. Infine abbiamo stampato un oggetto di tipo `ProgrammazioneFilm`, risultante dall'applicazione dell'`UnaryOperator` composto `ritardaDiUnOraEUnGiorno` sull'oggetto `shining`. L'output è il seguente:

```
Shining verrà proiettato il 2014-05-30 alle 22:00
```

Il vantaggio di aver concatenato in un unico `UnaryOperator` le due modifiche all'oggetto `ProgrammazioneFilm`, ci ha permesso di non fare passaggi intermedi ed ottenere direttamente lo scopo desiderato.

Riepilogo

In questo modulo abbiamo visto come Java 8 ha introdotto le **espressioni lambda** e i **reference a metodi**. La sintassi delle **espressioni lambda** è progettata per scrivere meno codice possibile, e lo stesso può dirsi per i **reference a metodi**. Abbiamo visto come questi due concetti siano intercambiabili e come il programmatore abbia la scelta di usare queste due sintassi, a seconda di quella che può risultare più comoda al momento. È fondamentale capire come le espressioni lambda (e quindi i **reference a metodi**) possano sostituire nella maggior parte dei casi quello che prima si faceva con una classe anonima. Abbiamo anche asserito più volte che un'espressione lambda sostituisce l'implementazione di un'interfaccia funzionale, ridefinendo il codice del suo metodo SAM. Le espressioni lambda essenzialmente forniscono oltre alla possibilità di passare codice ad un metodo (cosa che si poteva fare già con una classe anonima) anche la sinteticità e la semplicità (visto che non bisogna tener conto di tutte le regole che esistono per le classi anonime). Inoltre è importante

comprendere che quando passiamo una espressione lambda ad un metodo, permettiamo a quel metodo di definire quando invocare il codice passato.

Per quanto riguarda i **reference a metodi** abbiamo introdotto le quattro diverse sintassi utilizzabili, spiegandone il significato mediante semplici esempi. Infine abbiamo introdotto le interfacce principali dei package `java.util.function`: `Predicate`, `Consumer`, `Supplier`, `Function` e `UnaryOperator`.

Esercizi modulo 15

Esercizio 15.a) Espressioni lambda e reference a metodi:

1. Con un'espressione lambda è possibile fare tutto quello che fa una classe anonima.
2. Un'espressione lambda può utilizzare in maniera thread-safe le variabili d'istanza della classe in cui è dichiarata.
3. Un'espressione lambda può utilizzare le variabili d'istanza della classe in cui è definita solo se dichiarate `final`.
4. La seguente espressione lambda è legale:

```
Consumer <String> c = ((x)->System.out.println(x));
```

5. La seguente espressione lambda è legale:

```
(x, y) -> System.out.println(x);  
        System.out.println(y);
```

6. Il seguente reference a metodo è legale:

```
System.out::println()
```

7. Il seguente statement è legale:

```
System.out.println(Math::random)
```

8. Il seguente statement è legale:

```
System.out::println(Math::random)
```

9. Il seguente statement è legale:

```
Supplier<Chitarra> chitarraSupplier = Chitarra::new;
```

10. Supponendo che la classe `Chitarra` abbia un costruttore senza parametri, allora il seguente codice è legale:

```
Chitarrista chitarrista = new Chitarrista();  
chitarrista.suonaChitarra(Chitarra::new);
```

Soluzioni esercizi modulo 15

Esercizio 15.a) Espressioni lambda e reference a metodi:

1. **Falso**, per esempio in una classe anonima possiamo definire anche variabili d'istanza.
2. **Falso**, cfr. paragrafo 15.1.3.2
3. **Falso**.
4. **Falso**, mancano quantomeno le parentesi graffe al blocco di codice.
5. **Falso**, le parentesi vicino al metodo `println` non fanno parte della sintassi.
6. **Falso**.
7. **Falso**.
8. **Falso**.
9. **Vero**.
10. **Vero**.

**Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere la sintassi e le modalità d'uso che ci offre la nuova feature delle espressioni lambda (unità 15.1)	<input type="checkbox"/>	
Saper decidere quando utilizzare un'espressione lambda o una classe anonima (unità 15.1)	<input type="checkbox"/>	
Saper gestire la dinamicità del passaggio di codice ad un metodo con un'espressione lambda (unità 15.1)	<input type="checkbox"/>	
Saper definire ed utilizzare tutte le 4 diverse sintassi dei reference a metodi (unità 15.2)	<input type="checkbox"/>	
Saper utilizzare le interfacce funzionali definite nel package <code>java.util.function</code> (unità 5.3)	<input type="checkbox"/>	

Note:

Collections Framework e Stream API

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Conoscere l'utilità del framework Collections (unità 16.1).
- ✓ Conoscere le principali interfacce del framework Collections e i metodi che espongono (unità 16.2, 16.3, 16.4, 16.5, 16.6).
- ✓ Conoscere le principali implementazioni del framework Collections e i metodi che espongono (unità 16.2, 16.3, 16.4, 16.5, 16.6).
- ✓ Saper utilizzare le classi d'utilità Arrays e Collections e i metodi che espongono (unità 16.7).
- ✓ Conoscere l'utilità della libreria Stream API (unità 16.8).
- ✓ Saper utilizzare stream e pipeline (unità 16.8).
- ✓ Saper utilizzare metodi di aggregazione, di riduzione e di parallelismo (unità 16.8).

In questo modulo introdurremo finalmente uno degli argomenti più importanti per il programmatore Java: **il framework Collections**. Questo argomento viene presentato solo ora per ragioni didattiche, non certo perché poco rilevante. È probabilmente impossibile trovare un programma Java (almeno che non sia particolarmente banale) che non fa uso in qualche modo di questo framework. Anche negli esempi che abbiamo presentato sino ad ora siamo stati costretti ad anticipare alcune delle fondamentali implementazioni di questo framework, come le interfacce `Map` e `List`, e le loro implementazioni `ArrayList` ed `HashMap`. Il framework Collections è in buona parte contenuto nel package `java.util`, ma si estende anche nel package `java.util.concurrent` con una serie di interfacce e classi che definiscono collezioni per gestire la concorrenza dei processi.

Con il termine “framework” in questo caso intendiamo un insieme di classi e di interfacce riutilizzabili ed estendibili. Questa definizione dovrebbe essere un po’ più complessa ma per i nostri scopi può andare bene.

Una delle novità più importanti di Java 8 è l'introduzione della libreria denominata **Stream API**. Questa libreria è essenzialmente basata sull'interfaccia `Stream`, definita nel package `java.util.stream`, e ci permetterà di utilizzare le collezioni in maniera più semplice ed efficiente. Gli stream sono stati progettati per essere utilizzati congiuntamente alle espressioni lambda e ai riferimenti a metodi, quindi è fondamentale che il lettore abbia già studiato il modulo precedente. Gli stream ci faranno apprezzare ancora di più l'introduzione delle espressioni lambda.

Il framework Collections è stato introdotto in Java nella versione 1.2, portando il linguaggio ad un livello molto più alto rispetto a prima. Ma già nella versione 1.0 esistevano delle collezioni, tanto che queste sono in qualche modo considerate facenti parte del framework. Stiamo parlando in particolare delle classi `Hashtable` e `Vector`, e dell'interfaccia `Enumeration`. Queste sono spiegate nell'appendice O on line e sono citate all'interno di questo modulo per fare alcuni confronti.

16.1 Introduzione al framework Collections

Il framework noto come Collections è costituito da una serie di classi, interfacce e algoritmi per gestire collezioni. Una **collezione** è definibile come un oggetto che raggruppa più elementi in una singola unità. I vantaggi di avere a disposizione questo framework per la programmazione sono tanti: possibilità di scrivere meno codice, incremento delle performance, interoperabilità tra classi non relazionate tra loro, riusabilità, algoritmi complessi già implementati a disposizione, maggiore qualità e semplicità del codice, estensibilità per creare nuove librerie.

Il framework è basato su otto interfacce principali che vengono poi estese da tante altre classi astratte e concrete. In Figura 16.1, un diagramma delle classi UML semplificato (cfr. appendice I per definizione di UML e appendice L per sintassi UML) illustra la gerarchia di queste interfacce che è costituita da due alberi diversi.

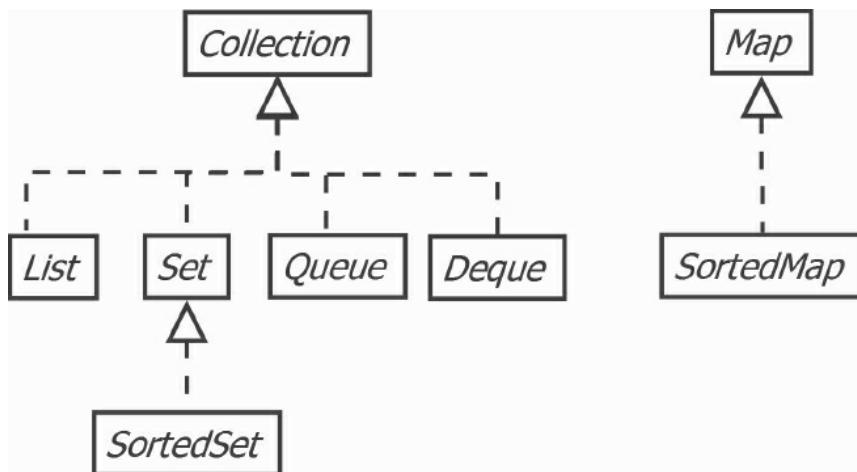


Figura 16.1 - La gerarchie tra le interfacce fondamentali del framework Collections.

Ad un primo livello si trovano le interfacce `Collection` (la più importante) e `Map`. `Collection` è estesa dalle interfacce `List`, `Queue`, `Deque` e `Set`. Quest'ultima a sua volta è estesa da `SortedSet`. `Map` è invece estesa da `SortedMap`. Ognuna di queste interfacce possiede proprietà che le sottoclassi ereditano. Per esempio, una classe che implementa `Set` (in italiano “insieme”) è un tipo di collezione non ordinata che non accetta elementi duplicati. Di seguito facciamo una breve panoramica su queste

interfacce e le loro proprietà.

L'interfaccia `Collection` è la radice principale di tutta la gerarchia del framework. Essa astrae il concetto di “insieme di oggetti”, detti **elementi**. Esistono implementazioni che ammettono elementi duplicati ed altre che non lo permettono, collezioni ordinate e non ordinate. La libreria non mette a disposizione alcuna implementazione diretta di `Collection`, ma solo delle sue dirette sottointerfacce come `Set` e `List`. L'interfaccia `Collection` viene definita come il *minimo comune denominatore* che tutte le collection devono implementare.

Un `Set` è un tipo di collection che, astraendo il concetto di insieme matematico, non ammette elementi duplicati.

Una `List` è una collezione ordinata (detta anche *sequence*). In una lista viene sempre associato un indice ad ogni elemento, che equivale alla posizione dell'elemento stesso all'interno della lista. Una lista ammette elementi duplicati (distinguibili fra di loro per la posizione).

L'interfaccia `Queue`, fornisce invece operazioni di inserimento, estrazione e ricerca, e gli elementi sono organizzati come in una pila FIFO (“First In First Out”).

L'interfaccia `Deque` (che è l'abbreviativo di “double ended queue” e si pronuncia “deck”) fornisce le stesse operazioni dell'interfaccia `Queue` ma utilizzabili ad entrambe le estremità della pila. I suoi elementi possono essere organizzati sia con gestione FIFO sia con LIFO (“Last In First Out”).

Una `Map` è una collezione che associa chiavi ai suoi elementi. Le mappe non possono contenere chiavi duplicate ed ogni chiave può essere associata ad un solo valore.

Le ultime due interfacce, `SortedSet` e `SortedMap`, rappresentano le versioni ordinate di `Set` e `Map`. Aggiungono diverse nuove funzionalità relative all'ordinamento, che avviene sempre tramite l'implementazione da parte degli elementi dell'interfaccia `Comparable`, oppure mediante la codifica di oggetti `Comparator` (cfr. paragrafo 11.1.9).

Attenzione che con Java 8 anche le interfacce possono definire metodi implementati o statici, e quindi troveremo metodi ereditati da queste interfacce già pronti per l'uso.

Pressoché tutte le classi e le interfacce di questo framework sono generiche, e se non usate specificando i tipi parametri, saranno considerate “raw type” dal compilatore, che quindi ci segnalera dei warning. Bisognerebbe quindi specificare sempre il tipo parametro per le collezioni. Possiamo affermare che i tipi generici sono usati soprattutto per le collezioni.

Andiamo quindi ad esplorare queste interfacce e le loro più importanti implementazioni nei prossimi paragrafi.

L'introduzione della libreria Stream API rivoluzionerà completamente il modo in cui usiamo le collezioni, pertanto introdurremo tale libreria nella maniera più pratica possibile.

16.2 L'interfaccia `Collection`

È la superinterfaccia per definizione del framework. Possiamo usare un reference di `Collection` per puntare a tutte le implementazioni che appartengono allo stesso albero gerarchico per il polimorfismo

per metodi. Per esempio possiamo scrivere:

```
Collection<String> collection = new ArrayList<>();
```

Questa pratica che sfrutta il polimorfismo per dati, è sempre consigliabile e ancora di più per le collezioni. Per esempio se avessimo scritto tante altre righe di un programma che utilizzano la variabile `collection`, e avessimo necessità di voler cambiare il tipo dell'oggetto per esempio con una `LinkedList`, l'unica riga da cambiare sarebbe proprio quella appena scritta. Tutte le altre parti di codice che usano `collection` continueranno a funzionare.

L'interfaccia `Collection` definisce metodi di base come:

- `int size()` ritorna il numero degli elementi contenuti nella collezione.
- `boolean add(E element)` aggiunge un elemento (del tipo parametro) alla collezione e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno.
- `boolean remove(Object element)` rimuove un elemento dalla collezione e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno.
- `boolean isEmpty()` restituisce `true` o `false` a seconda del fatto che la collezione contenga elementi o meno.
- `void clear()` elimina tutti gli elementi dalla collezione.
- `boolean contains(Object element)` restituisce `true` o `false` a seconda del fatto che l'elemento specificato in input sia contenuto nella collezione o meno. Per verificare l'uguaglianza viene utilizzato il metodo `equals()`.
- `Iterator<E> iterator()` restituisce un'implementazione di un'interfaccia `Iterator` che serve per iterare sugli elementi della collezione (cfr. prossimi paragrafi). Per esempio il seguente frammento di codice:

```
Collection<String> progBands = new HashSet<>();
progBands.add("Dream Theater");
progBands.add("Ayreon");
progBands.add("Yes");
Iterator<String> iterator = progBands.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

stamperà:

```
Ayreon
Dream Theater
Yes
```

Si noti che l'`HashSet` (come tutte gli oggetti `Set` che non implementano `SortedSet`) non mantiene l'ordine in cui sono stati inseriti gli elementi.

Inoltre `Collection` definisce metodi per interagire con altre collezioni (ovvero con implementazioni di `List`, `Set` e così via):

- ❑ `boolean containsAll(Collection<?> c)` restituisce `true` o `false` a seconda del fatto che gli elementi della collezione specificata in input siano tutti o meno contenuti nella collezione su cui viene chiamato il metodo. Questo indipendentemente dall'ordine degli oggetti nelle collezioni.

Per esempio il seguente frammento di codice:

```
Collection<Integer> collection1 = new ArrayList<>();  
for (int i = 0; i < 5; i++) {  
    collection1.add(i);  
}  
Collection<Integer> collection2 = new ArrayList<>();  
collection2.add(2);  
collection2.add(1);  
collection2.add(4);  
System.out.println(collection1.containsAll(collection2));
```

stamperà:

```
true
```

- ❑ `boolean addAll(Collection<? extends E> c)` aggiunge tutti gli elementi della collezione specificata in input alla collezione su cui è stato chiamato il metodo, e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno.
- ❑ `boolean removeAll(Collection<?> c)` rimuove tutti gli elementi della collezione specificata in input, dalla collezione su cui è stato chiamato il metodo, e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno.
- ❑ `boolean retainAll(Collection<?> c)` per la collezione su cui è stato chiamato il metodo conserva solamente gli elementi della collezione specificata in input, rimuovendo gli altri, e restituisce `true` o `false` a seconda del fatto che l'operazione sia andata a buon fine o meno.
Per esempio, considerando le due collezioni inizializzate come nell'esercizio precedente, allora il seguente frammento di codice:

```
collection1.retainAll(collection2);  
System.out.println(collection1);
```

stamperà:

```
[1, 2, 4]
```

Si noti che per questa stampa è stato invocato automaticamente il metodo `toString()` di `ArrayList` che ha formattato gli elementi della collezione in modo leggibile.

L'interfaccia `Collection` definisce anche metodi per interagire con array:

- `Object[] toArray()` restituisce un array di `Object` a partire da una collezione.
- `<T> T[] toArray(T[] a)` restituisce un array di oggetti di tipo parametro `T` a partire da una collezione. Per esempio considerando l'oggetto `progBands` dell'esempio sul metodo `iterator()`, con il seguente codice andiamo a riempire un array di stringhe con il contenuto della collezione, e il successivo ciclo `foreach` produrrà lo stesso output del ciclo che abbiamo visto prima con l'`iterator`:

```
String [] progBandsArray = progBands.toArray(new String[0]);  
for (String progBand : progBandsArray) {  
    System.out.println(progBand);  
}
```

È possibile anche il processo inverso, utile per riempire una collezione (in particolare una lista) a partire da un array (cfr. metodo `asList()` della classe di utilità `Arrays`).

Infine, la novità importante che ha portato Java 8, è che questa interfaccia espone i metodi `stream()` e `parallelStream()`. Questi restituiscono oggetti di tipo `Stream` che consentono di eseguire cicli in maniera sequenziale o parallela sulla collezione. Gli stream si fanno preferire, rispetto agli iteratori e ai costrutti `foreach`, per le loro performance e il loro supporto alle espressioni lambda. Quindi gli stream permettono di scrivere meno codice con più efficienza. Approfondiremo il discorso sugli stream nella seconda parte di questo modulo.

16.2.1 Iterare sulle collezioni

Esistono tre modi per iterare sulle collezioni:

1. Usando cicli `foreach`.
2. Usando iteratori.
3. Usando il metodo di default `forEach()` definito nell'interfaccia `Collection` e sfruttando le cosiddette **operazioni di aggregazione** (in inglese **aggregate operations**) con la nuova libreria **Stream API**.

Questo vale per tutte le collezioni, tuttavia implementazioni più specifiche possono utilizzare anche altre tipologie di iterazioni come cicli `for` ordinari sfruttando un indice (è il caso delle liste).

16.2.1.1 Ciclo `foreach`

L'interfaccia `Collection` estende l'interfaccia `Iterable`, e tutto quello che estende `Iterable` può essere iterato tramite un ciclo `foreach`. Per esempio considerando sempre la collezione `progBands` già usata nei precedenti esempi, possiamo iterare su di essa anche in questo modo:

```
for (String progBand : progBands) {  
    System.out.println(progBand);  
}
```

Bisognerebbe utilizzare questo tipo di costrutto quando è necessario iterare su collezioni di piccole dimensioni e non si devono effettuare operazioni impegnative sugli elementi.

Nell'appendice O on line è descritto anche come creare propri tipi implementando **Iterable**.

16.2.1.2 Iteratori

Gli **iteratori** sono oggetti che astraggono dei cursori per eseguire cicli sugli elementi di una collezione. Essi possono spostarsi elemento per elemento (uno alla volta) partendo dal primo e arrivando all'ultimo, e in generale non possono cambiare direzione e tornare indietro. L'interfaccia **Iterator** come abbiamo già visto, viene usata sfruttando il metodo `iterator()` di `Collection`. Infatti ogni implementazione ha la sua istanza di `Iterator`. Per esempio, sempre considerando l'oggetto `progBands` degli esempi precedenti, la seguente istruzione:

```
Iterator<String> iterator = progBands.iterator();
```

stamperà:

```
java.util.HashMap$KeyIterator
```

Ovvero, tramite il metodo `getName()` chiamato sull'oggetto `Class` che ci ha ritornato il metodo `getClass()`, abbiamo scoperto che stiamo usando un oggetto istanziato da una classe innestata di `HashMap` che si chiama `KeyIterator`. Fortunatamente grazie all'astrazione fornita dal polimorfismo per dati, non ci interessa conoscere questa implementazione. Semplicemente ci limiteremo sempre ad utilizzare i metodi che appartengono all'interfaccia `Iterator`.

Per cultura e completezza è utile sapere che, nonostante possa sembrare strano al neofita adoperare un iteratore per eseguire un ciclo su una collezione (il ciclo `foreach` è sicuramente più intuitivo e semplice da utilizzare), "Iterator" è anche il nome di un famoso Design Pattern sul cui modello è stata creata quest'interfaccia. Come tanti altri pattern questa soluzione si basa su uno dei principi fondamentali dell'Object Orientation: il "Dependency Inversion Principle" (conosciuto anche con il suo acronimo "DIP") che in italiano si traduce come "Principio di inversione della dipendenza". Questo principio ben formulato da Robert C. Martin ci consiglia di far dipendere le nostre classi dalle astrazioni e non dai dettagli concreti delle implementazioni. Essenzialmente è il principio che ispira il polimorfismo stesso. Nell'esempio

dell'iteratore è possibile apprezzare pienamente l'efficacia di tale convenzione, facendo dipendere l'iteratore dall'astrazione (interfaccia) `Iterator`, e andando ad ignorare completamente l'implementazione della classe interna `KeyIterator` di `HashMap`.

L'interfaccia `Iterator` definisce solo tre metodi molto semplici. Segue la sua dichiarazione epurata di tutto quello che al momento non serve:

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

Il metodo `hasNext()`, già usato nell'esempio in cui è stato presentato il metodo `iterator()` di `Collection`, restituisce `true` se l'iteratore non si trova sull'ultimo elemento (e quindi dopo non ci sono più elementi). Il metodo `next()` lo abbiamo visto far avanzare di un passo l'iteratore. Infine il metodo `remove()` rimuove l'elemento corrente (ovvero dove si trova l'iteratore). È molto importante sapere che questo metodo è l'unico che permette la rimozione di un elemento da una collezione durante un'iterazione in maniera thread-safe. Ogni altro tentativo non è sicuro. Ecco un esempio banale dove con un iteratore rimuoviamo un elemento della collezione `progBands`:

```
Iterator<String> iterator = progBands.iterator();  
while (iterator.hasNext()) {  
    if (iterator.next().equals("Ayreon")) {  
        iterator.remove();  
    }  
}  
System.out.println(progBands);
```

l'output sarà:

```
[Dream Theater, Yes]
```

Esiste anche un quarto metodo introdotto con Java 8. Il metodo `forEachRemaining()` prende in input un oggetto di tipo `Consumer` (cfr. paragrafo 15.3.2). Questo è un metodo di default, quindi ha già un'implementazione, ma questo ci interessa relativamente. Esso esegue l'operazione specificata per tutti gli elementi su cui bisogna ancora eseguire il ciclo, fino a quando sono finiti oppure se scatta un'eccezione. Per esempio, consideriamo la seguente classe `Smartphone`:

```
public class Smartphone {  
    private String modello;  
    private String marca;  
    private int prezzo;  
    // Costruttori e metodi set e get omessi  
    public String toString() {
```

```
        return marca + " " + modello;
    }
}
```

Il seguente codice:

```
Collection<Smartphone> smartphones = new HashSet<>();
smartphones.add(new Smartphone("Samsung", "Note 74"));
smartphones.add(new Smartphone("Apple", "Iphone 225"));
smartphones.add(new Smartphone("HTC", "One M167"));
Iterator<Smartphone> iterator = smartphones.iterator();
iterator.forEachRemaining(element-> {element.setMarca("Google");});
System.out.println(smartphones);
```

stamperà:

```
[Google One M167, Google Note 74, Google Iphone 225]
```

Si noti che abbiamo modificato la marca di tutti gli smartphone con un'unica chiamata al metodo `forEachRemaining()` ed usando una semplice espressione lambda.

In generale bisognerebbe usare un `Iterator` in luogo di un ciclo `foreach` quando c'è bisogno di rimuovere elementi dalla collection (il ciclo `foreach` non mette a disposizione un `Iterator`, e di conseguenza non c'è metodo `remove()` da invocare).

16.2.1.3 Metodo `forEach()` dell'interfaccia `Iterable`

Con l'avvento di Java 8, l'interfaccia `Iterable` si arricchisce del metodo di default `forEach()` (da non confondere con il costrutto che di solito chiamiamo `foreach`). Il metodo `forEach()` prende in input un oggetto di tipo `Consumer` (cfr. paragrafo 15.3.2) che ricordiamo, ci permette di eseguire operazioni su oggetti (di solito per modificarne lo stato interno). Questo significa che qualsiasi collezione potrà iterare direttamente sui suoi elementi semplicemente chiamando questo metodo, e passandogli in input un'espressione lambda per fare operazioni sugli stessi elementi. Per esempio, considerando l'oggetto `smartphones` dell'esempio precedente, con la seguente istruzione:

```
smartphones.forEach(s->System.out.println(s));
```

saranno stampati tutti gli elementi della collezione. Infatti, il metodo `forEach()` eseguirà l'espressione lambda (in questo caso un'operazione di stampa) per ogni elemento della collezione.

16.3 Interfaccia `List`

L'interfaccia `List` rappresenta una collezione ordinata e indicizzata di elementi in cui sono ammessi duplicati. Eredita tutti i metodi che vengono ereditati dall'interfaccia `Collection`. In generale una lista definisce i metodi `add()` e `addAll()` (che ricordiamo prende in input una collezione) in modo

tale che aggiungano in coda gli elementi in input. Quindi la sequenza di come gli elementi di una lista sono aggiunti, ne definisce l'ordine.

Uno dei metodi più utilizzati in assoluto è il metodo `get()` che prende in input un indice intero e restituisce l'oggetto corrispondente a quell'indice. Si usa per esempio quando si itera su una lista con un ciclo `for` sfruttando l'indice della collezione:

```
List<Integer> list = new ArrayList<>(3);
list.add(25);
list.add(7);
list.add(74);
int size = list.size();
for (int i = 0; i < size; i++) {
    System.out.println(list.get(i));
}
```

Esiste anche il metodo `add()` che prende in input non solo l'elemento da aggiungere ma anche l'indice della posizione dove deve essere inserito, eseguendo uno shift in avanti di tutti gli altri elementi, se ve ne sono. Equivalentemente è definito il metodo `set()` che prende in input sia l'elemento da impostare che l'indice della posizione dove deve essere impostato, ma questa volta l'elemento in input va semplicemente a sovrascrivere l'eventuale elemento già esistente a quell'indice.

Interessante anche la possibilità di ottenere, mediante la chiamata al metodo `listIterator()`, un oggetto di tipo `ListIterator` che a differenza di un `Iterator` classico, permette di iterare in entrambe le direzioni. Infatti definisce metodi come `hasPrevious()` e `previous()` che sono gli equivalenti di `hasNext()` e `next()` ma si riferiscono all'elemento precedente invece che al successivo.

16.3.1 Implementazioni di List

Le principali implementazioni di `List` sono `ArrayList`, e `LinkedList`.

`ArrayList` è in assoluto la collezione più utilizzata ed è stata creata proprio per rappresentare una evoluzione ridimensionabile di un array, sostituendo la superata classe `Vector` (cfr. appendice O on line). `Vector` e `ArrayList` hanno le stesse funzionalità, ma quest'ultima è più performante perché non è sincronizzata. `ArrayList` ha prestazioni nettamente superiori anche a `LinkedList`, che conviene utilizzare solo per gestire collezioni di tipo "coda". Si tratta di una cosiddetta lista concatenata, nella quale ogni elemento mantiene un reference verso l'elemento successivo e l'elemento precedente. Mette a disposizione tra l'altro metodi come `addFirst()`, `getFirst()`, `removeFirst()`, `addLast()`, `getLast()` e `removeLast()`. È quindi opportuno scegliere l'utilizzo di una `LinkedList` in luogo di `ArrayList` solo quando si devono aggiungere spesso elementi all'inizio della lista, oppure quando si vogliono eliminare elementi all'interno della lista durante le iterazioni. Queste operazioni richiedono infatti un tempo costante nelle `LinkedList` e un tempo lineare (ovvero che dipende dal numero degli elementi) in un `ArrayList`. In compenso però, l'accesso posizionale in una `LinkedList` è lineare, mentre è costante in un `ArrayList`. Questo implica una prestazione superiore da parte dell'`ArrayList` nella maggior parte dei casi.

La classe `LinkedList` implementa anche l'interfaccia `queue`, una delle interfacce base (introdotta nella versione 1.5), e l'interfaccia `Deque` (introdotta dalla versione 1.6). `Deque` è un termine abbreviativo per definire una “double ended queue”. I metodi `addFirst()`, `getFirst()`, `removeFirst()`, `addLast()`, `getLast()` e `removeLast()` di cui sopra non sono altro che implementazioni dei metodi astratti di `Deque`.

Anche l'`ArrayList` possiede un parametro di configurazione: la capacità iniziale. Se istanziamo un `ArrayList` con capacità iniziale 20, avremo un oggetto con 20 posizioni vuote (ovvero in ogni posizione c'è un reference che punta a `null`) pronte a essere riempite. Quando sarà aggiunto il ventunesimo elemento, l'`ArrayList` si ridimensionerà automaticamente per avere capacità ventuno, e questo avverrà per ogni nuovo elemento. Per ogni nuovo elemento che si vuole aggiungere oltre la capacità iniziale, l'`ArrayList` dovrà prima ridimensionarsi e poi aggiungere l'elemento. Questa doppia operazione porterà ad un decadimento delle prestazioni.

È però possibile ottimizzare la prestazione di una `ArrayList` nel caso si vogliano aggiungere nuovi elementi superata la capacità iniziale. Infatti, quest'ultima si può modificare a piacimento “al volo”, in modo tale che l'`ArrayList` non sia costretto a ridimensionarsi per ogni nuovo elemento. Per fare ciò, basta utilizzare il metodo `ensureCapacity()` passandogli la nuova capacità, prima di chiamare il metodo `add()`. Per avere un'idea di quanto sia importante ottimizzare un `ArrayList`, viene presentato un semplice esempio. Sfruttiamo il metodo statico `currentTimeMillis()` della classe `System` (cfr. paragrafo 11.1.3) per calcolare i millisecondi che impiega un ciclo a riempire l'`ArrayList`:

```
//capacità iniziale 1
ArrayList<String> list = new ArrayList<>(1);
long startTime = System.currentTimeMillis();
list.ensureCapacity(100000000);
for (int i = 0; i < 100000000; i++) {
    list.add("nuovo elemento");
}
long endTime = System.currentTimeMillis();
System.out.println("Tempo = " + (endTime - startTime));
```

L'output sul mio portatile è:

```
Tempo = 347
```

Commentando la riga:

```
list.ensureCapacity(100000000);
```

l'output cambierà:

```
Tempo = 1208
```

La performance è nettamente peggiore e la differenza sarà ancora più evidente aumentando il numero di elementi.

Se rimuoviamo un elemento da un `ArrayList`, la sua capacità non diminuisce. Esiste il metodo `trimToSize()` per ridurre la capacità dell'`ArrayList` al numero degli elementi effettivi.

16.4 Le interfacce `Set` e `SortedSet`

L'interfaccia `Set` rappresenta una collezione non ordinata di elementi in cui non sono ammessi duplicati. In particolare l'interfaccia `Set` non definisce altri metodi rispetto a `Collection`, bensì impone regole più restrittive quando questi vengono utilizzati. Questo significa che se per esempio aggiungiamo due volte tramite il metodo `add()` lo stesso elemento, questo verrà aggiunto solamente una volta.

L'interfaccia `SortedSet` invece rappresenta una collezione ordinata di elementi in cui non sono ammessi duplicati. Questa definisce alcuni metodi per lavorare su range di elementi, o sugli estremi della collezione.

16.4.1 Implementazioni di `Set` e `SortedSet`

Un'implementazione di `Set` è `HashSet`, mentre un'implementazione di `SortedSet` è `TreeSet`. `HashSet` è più performante rispetto a `TreeSet` ma non gestisce l'ordinamento. Entrambe queste classi non ammettono elementi duplicati e sono di sicuro le implementazioni più utilizzate di insiemi. `HashSet` garantisce performance migliori e dovrebbe essere sempre preferita a `TreeSet`, a meno che non si voglia iterare in maniera ordinata sugli elementi della collezione.

Segue un esempio di utilizzo di `TreeSet` dove si aggiungono elementi di tipo stringa e poi si sfrutta il metodo `forEach()` con un reference al metodo `println()` di `System.out` per stamparli:

```
Set<String> set = new TreeSet<>();
set.add("c");
set.add("a");
set.add("b");
set.add("b");
set.forEach(System.out::println);
```

L'output sarà:

```
a
b
c
```

Infatti l'elemento duplicato (`b`) non è stato aggiunto e gli elementi sono stati ordinati secondo la loro natura di stringhe. Per ordinare la collezione è stato utilizzato il metodo `compareTo()` dell'interfaccia `Comparable` implementata da `String`. Ma esiste anche un costruttore di `TreeSet` che prende in input un `Comparator`, per far sì che la collezione sia ordinata mediante un metodo personalizzato. Se istanziassimo l'oggetto `TreeSet` nel seguente modo passandogli un'espressione

lambda che inverte l'algoritmo di ordinamento naturale:

```
TreeSet<String> set = new TreeSet<>((x,y) -> { return -(x.compareTo(y)); });
```

otterremmo l'output:

```
c  
b  
a
```

Dalla versione 1.6 gli oggetti `TreeSet` sono anche bidirezionali. Infatti, è anche possibile ottenere un'istanza di `Iterator` che itera gli elementi del `TreeSet` dall'ultimo al primo, invocando il metodo `descendingIterator()`:

```
Iterator iter = set.descendingIterator();  
while (iter.hasNext()) {  
    System.out.println(iter.next());  
}
```

Come è possibile immaginare se avessimo istanziato un `HashSet` in luogo di `TreeSet`, come di seguito:

```
Set<String> set = new HashSet<>();
```

avremmo ottenuto un ordinamento non prevedibile.

Un'altra implementazione degna di nota è `LinkedHashSet` introdotta nella versione 1.4. È una collezione che può essere considerata un po' un `Hashtable` e un po' una `LinkedList`. Rispetto ad un `HashSet` questa mantiene l'ordine degli elementi compatibile con il loro ordine di inserimento, come fanno le liste. Se nell'esempio precedente avessimo istanziato un `LinkedHashSet` in luogo di un `TreeSet`:

```
Set<String> set = new LinkedHashSet<>();
```

avremmo ottenuto il seguente output:

```
c  
a  
b
```

È interessante notare nella versione 6 di Java sia stata introdotta una nuova interfaccia che estende `SortedSet`: `NavigableSet`. Il discorso è molto simile a quello che faremo per `NavigableMap`. Quest'interfaccia definisce nuovi metodi per navigare la collezione come `lower()`, `floor()`, `ceiling()` e `higher()`, che restituiscono rispettivamente un elemento minore, minore o uguale, maggiore o uguale e maggiore dell'elemento specificato come argomento. La classe `TreeSet` implementa proprio quest'interfaccia e il metodo `descendingIterator()` è anch'esso definito

nell'interfaccia `NavigableSet`.

16.5 Le interfacce `Queue` e `Deque`

L'interfaccia `Queue` (in italiano “coda”) rappresenta una collezione progettata per raccogliere elementi prima di essere processata. Estende l'interfaccia `Collection` definendo nuovi metodi per l'inserimento, la rimozione e l'utilizzo dei dati. Ognuno di questi metodi è presente in due formati differenti: se l'operazione fallisce un formato lancia un'eccezione mentre l'altro restituisce un valore speciale (per esempio `null` o `false`). In particolare quando parliamo di valore speciale ci riferiamo alla situazione in cui il metodo restituisce o l'oggetto stesso appena aggiunto o recuperato, oppure un booleano (come nel caso del metodo `offer()`). Quindi, a seconda dell'esigenza, lo sviluppatore può usufruire di un metodo piuttosto che di un altro. La seguente tabella riassume quanto appena asserito.

Tipo di operazione	Metodo che lancia un'eccezione	Metodo che ritorna un valore speciale
Inserimento	<code>add(e)</code>	<code>offer(e)</code>
Rimozione	<code>remove()</code>	<code>poll()</code>
Recupero	<code>element()</code>	<code>peek()</code>

Le collezioni sono tutte ridimensionabili e quindi in generale l'inserimento è sempre possibile. L'operazione potrebbe fallire però nel caso di implementazioni di `Queue` con dimensione limitata. In particolare il metodo `offer()` inserisce un elemento ritornando `true` o `false` qualora l'operazione di inserimento riesca oppure no. La differenza essenziale con il metodo `add()` (già definito nell'interfaccia `Collection`) è che quest'ultimo può fallire nell'aggiungere un elemento solo lanciando una `unchecked exception`. Il metodo `offer()` è invece progettato per essere utilizzato quando il fallimento di un inserimento non rappresenta un evento eccezionale, per esempio proprio nel caso di code con dimensione massima.

I metodi `remove()` e `poll()` ritornano e rimuovono l'elemento che si trova in testa alla coda. Nel caso in cui non ci sia niente da rimuovere nella coda, il metodo `poll()` ritorna `null`, mentre `remove()` lancia un'eccezione. Il metodo `poll()` ritorna un riferimento all'oggetto rimosso in caso di successo.

I metodi `element()` e `peek()` invece ritornano ma non rimuovono l'elemento che si trova in testa alla coda. Nel caso in cui non ci sia niente da rimuovere nella coda, il metodo `peek()` ritorna `null`, mentre `element()` lancia un'eccezione. Il metodo `peek()` ritorna un riferimento all'oggetto rimosso in caso di successo.

La “testa della coda” è definita dall'implementazione della coda. Esistono code LIFO (che sta per “Last In First Out”) che definiscono come testa della coda il primo elemento inserito. Un'implementazione di coda LIFO l'abbiamo già vista: la classe `LinkedList` che mette a disposizione i metodi `addLast()`, `getLast()` e `removeLast()`. In aggiunta, `LinkedList` implementa pure l'interfaccia `Deque` e quindi può essere utilizzata come coda FIFO (che sta per “First In First Out”) dove la testa della coda è il primo elemento inserito. Per questo motivo, `LinkedList` mette a disposizione anche i metodi `addFirst()`, `getFirst()` e `removeFirst()`.

L'interfaccia `Deque` (che si pronuncia “deck”) è una collezione lineare che supporta l'inserimento e la rimozione dei suoi elementi ad entrambe le estremità. Può essere quindi utilizzata come una coda FIFO o LIFO a seconda della necessità. Come per `Queue`, le funzionalità di questa interfaccia sono

duplicate per lanciare eccezioni o ritornare valori speciali. La seguente tabella riassume i principali metodi definiti in `Deque`.

Tipo di operazione	Metodo che lancia un'eccezione	Metodo che ritorna un valore speciale	Metodo bloccante indefinito	Metodo bloccante con time out
Inserimento	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e, time, unit)</code>
Rimozione	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time, unit)</code>
Recupero	<code>element()</code>	<code>peek()</code>		

16.5.1 L'interfaccia `BlockingQueue` e implementazioni di `Queue` e `Deque`

Un'implementazione di `Queue` è definita dalla classe `PriorityQueue`, che ordina i propri elementi a seconda del proprio ordinamento naturale (definito mediante l'implementazione dell'interfaccia `Comparable`) o a seconda di un oggetto `Comparator` associato al momento della creazione. Attenzione che usando un `Iterator` per iterare su di essa, non è garantito che i suoi elementi vengano iterati nell'ordine che ci si aspetta. Infatti per ragioni prestazionali, i suoi elementi sono gestiti in background senza rispettare l'ordinamento, utilizzando una lista. Il consiglio per iterare gli elementi ordinati è quello di usare un'istruzione come la seguente:

```
Arrays.sort(pq.toArray());
```

Dove `pq` è un reference a un oggetto `PriorityQueue`. Infatti abbiamo prima trasformato in array la priority queue, e poi ordinato i suoi elementi mediante il metodo statico `sort()` della classe di utilità `Arrays` (che tratteremo più avanti).

Nel package `java.util.concurrent` è definita l'interfaccia `BlockingQueue`, la quale estende `Queue` e definisce nuovi metodi che bloccano con un thread apposito il recupero e la rimozione di un elemento fino a quando la coda diventa non vuota, e bloccano un inserimento fino a quando lo spazio nella coda bloccante diventa disponibile. Infatti è possibile limitare la capacità di una `BlockingQueue`, di solito mediante un costruttore apposito (come accade per l'implementazione `ArrayBlockingQueue`).

Se non si limita esplicitamente la capacità di una `BlockingQueue`, la capacità massima sarà pari al valore di `Integer.MAX_VALUE`, ovvero il più grande numero intero. In questo caso si parla di "unbounded queue" (o coda illimitata).

Oltre ai metodi che vengono ereditati dall'interfaccia `Queue`, `BlockingQueue` definisce altre due categorie di metodi, che appunto bloccano l'inserimento e la rimozione degli elementi secondo

quanto detto prima, oppure bloccano queste operazioni per un tempo massimo specificato (time-out). Nel caso il tempo specificato passi prima che l'operazione di inserimento o rimozione sia possibile, viene restituito un valore booleano `false`. Segue una tabella esplicativa.

Tipo di operazione	Metodo che lancia un'eccezione	Metodo che ritorna un valore speciale	Metodo bloccante indefinito	Metodo bloccante con time out
Inserimento	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e, time, unit)</code>
Rimozione	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time, unit)</code>
Recupero	<code>element()</code>	<code>peek()</code>		

Per i metodi che gestiscono il timeout bisogna specificare come altro parametro un `long` (`time` nella tabella) che rappresenta il tempo massimo per eseguire l'operazione. L'unità di tempo viene però specificata con il terzo parametro (`unit` nella tabella) che è di tipo `TimeUnit`. Si tratta di un'enumerazione che definisce come suoi elementi unità temporali: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `NANOSECONDS`, `SECONDS` e `MINUTES`.

Le implementazioni dell'interfaccia `BlockingQueue`, sono diverse:

- ❑ `LinkedBlockingQueue`, una coda FIFO illimitata (ma che può essere limitata tramite costruttore) e bloccante che in background usa `LinkedList`.
- ❑ `PriorityBlockingQueue`, una `PriorityQueue` illimitata e bloccante.
- ❑ `ArrayBlockingQueue`, una coda limitata che usa in background array.
- ❑ `DelayQueue`, una coda illimitata nella quale un suo elemento è reso disponibile solo dopo che un certo timeout configurato è scaduto.
- ❑ `SynchronousQueue`, una coda bloccante nella quale ogni operazione di inserimento deve aspettare la corrispondente operazione di rimozione da un altro thread e viceversa.
- ❑ `LinkedTransferQueue`, implementa la sottointerfaccia di `BlockingQueue`, `TransferQueue`, introdotta con Java 7. Questa permette al codice che aggiunge un elemento alla coda, di avere l'opzione per bloccare il codice di un altro thread che vuole usare quell'elemento.

Per quanto riguarda le implementazioni di `Deque`, esse sono molto simili a quelle di `Queue` tranne per il fatto che implementano `Deque`. La più famosa come abbiamo detto è `LinkedList`. Altre implementazioni da menzionare sono:

- ❑ `ArrayList`, è la versione ridimensionabile di `LinkedList` basata in background su array (mentre `LinkedList` usa `List`). Essa è più performante di `LinkedList`, consuma meno memoria e non ammette elementi `null`.
- ❑ `LinkedBlockingQueue`, fa parte del package `java.util.concurrent` ed è la versione bloccante della `Deque`. Se la `Deque` è vuota, allora una chiamata a `takeFirst()` o `takeLast()`

aspetterà che l'elemento sia disponibile, quindi otterrà un reference e rimuoverà lo stesso elemento.

16.6 Le interfacce Map e SortedMap

Una mappa è un oggetto che mappa chiavi a valori. Una mappa non può contenere chiavi duplicate, ogni mappa può contenere al massimo un solo valore. È definita usando due tipi parametro, una per le chiavi (`K`, iniziale di “key” che in inglese significa “chiave”) e una per i valori (`V`, iniziale di “value” che in inglese significa “valore”):

```
public interface Map<K,V> { // . . . }
```

Seguono i metodi fondamentali definiti da questa interfaccia:

- ❑ `V put(K key, V value)` aggiunge una coppia chiave-valore alla mappa e ritorna il vecchio valore associato alla chiave (se era presente) oppure `null`.
- ❑ `V get(Object key)` restituisce il valore associato alla chiave passata in input. Potrebbe tornare `null`.
- ❑ `V remove(Object key)` rimuove la mappatura chiave-valore associata alla chiave specificata e ritorna il vecchio valore associato alla chiave (se era presente) oppure `null`.
- ❑ `void putAll(Map<? extends K, ? extends V> m)` aggiunge tutte le mappature della mappa passata in input alla mappa corrente.
- ❑ `boolean containsKey(Object key)` restituisce `true` o `false` a seconda se la chiave specificata è presente.
- ❑ `boolean containsValue(Object value)` restituisce `true` o `false` a seconda se il valore specificato è presente o meno.
- ❑ `Set<K> keySet()` ritorna un `Set` contenente tutte le chiavi presenti nella mappa.
- ❑ `Collection<V> values()` ritorna una `Collection` di tutti i valori presenti nella mappa.
- ❑ `Set<Map.Entry<K, V>> entrySet()` ritorna un `Set` contenente oggetti di tipo `Map.Entry` (ovvero di tipo `Entry` interfaccia interna di `Map`) che rappresentano le corrispondenze chiave-valore presenti. Questi oggetti mettono a disposizione metodi come `getKey()` e `getValue()` per recuperarne rispettivamente la chiave e il valore.

Inoltre sono definiti diversi metodi di default tra cui il metodo `forEach()`.

L'interfaccia `SortedMap` è la versione ordinata di `Map`, come `SortedSet` lo era di `Set`. L'ordinamento è eseguito sulle chiavi. Definisce nuovi metodi come i seguenti:

- ❑ `K firstKey()` restituisce la prima chiave della mappa.
- ❑ `K lastKey()` restituisce l'ultima chiave della mappa.
- ❑ `Comparator<? super K> comparator()` restituisce l'oggetto `Comparator` utilizzato per ordinare gli elementi della mappa.

- ❑ `SortedMap<K, V> subMap(K fromKey, K toKey)` restituisce il sottoinsieme della mappa compreso tra le chiavi specificate `fromKey` e `toKey`. Questo è possibile solo perché `SortedSet` è ordinata.
- ❑ `SortedMap<K, V> headMap(K toKey)` restituisce il sottoinsieme della mappa compreso tra la prima chiave e quella specificata `toKey`.
- ❑ `SortedMap<K, V> tailMap(K fromKey)` restituisce il sottoinsieme della mappa compreso tra la chiave specificata `fromKey` e l'ultima.

16.6.1 Implementazioni di Map e SortedMap

Una classe non sincronizzata (e quindi ancora più performante) e del tutto simile ad `Hashtable` (cfr. appendice O on line) è la classe `HashMap`. A meno che non ci sia la necessità di avere una mappa che gestisca la concorrenza in scrittura, `HashMap` dovrebbe sempre essere preferita a `Hashtable`.

Per quanto riguarda le classi `Hashtable` e `HashMap` esistono regole per gestirne al meglio la prestazione che si basano su parametri detti “load factor”, o fattore di carico, e “buckets” (che rappresentano la capacità). È un argomento complesso e per ora trascurabile, ma chi volesse approfondire può consultare la documentazione ufficiale.

Un'altra classe molto simile, ma questa volta implementazione di `SortedMap`, è invece `TreeMap`. La classe `HashMap` è notevolmente più performante rispetto a `TreeMap`. Quest'ultima però gestisce l'ordinamento, che non è una caratteristica di tutte le mappe. L'ordinamento è gestito sulle chiavi e non sui valori. Solamente dalla versione 1.6 `TreeMap` è stata rivista per implementare una nuova interfaccia: `NavigableMap`. Questa estende `SortedMap` aggiungendo nuovi metodi come `lowerEntry()`, `floorEntry()`, `ceilingEntry()` e `higherEntry()`, che restituiscono oggetti di tipo `Map.Entry` associati con le chiavi rispettivamente minore, minore o uguale, maggiore o uguale e maggiore della chiave specificata come argomento.

Ricordiamo che l'interfaccia innestata `Map.Entry` astrae una coppia di tipo chiave-valore che rappresenta un elemento di una mappa.

Inoltre vengono definiti anche i metodi `firstEntry()`, `pollFirstEntry()`, `lastEntry()` e `pollLastEntry()` che restituiscono e/o rimuovono il primo o l'ultimo (secondo l'ordinamento) oggetto `Map.Entry` della mappa.

Ai metodi `keySet()` ed `entrySet()` ereditati da `SortedMap`, che restituiscono rispettivamente un insieme ordinato (`SortedSet`) delle chiavi o degli elementi della mappa, vengono aggiunti i metodi `descendingKeySet()` e `descendingEntrySet()`. Questi restituiscono rispettivamente un insieme (`SortedSet`) delle chiavi o degli elementi della mappa, questa volta ordinato al contrario, dall'ultimo elemento al primo.

I metodi `entrySet()` e `descendingEntrySet()` restituiscono un insieme di valori ordinati per chiave e non per valore.

Esiste anche la classe `LinkedHashMap`, la versione concatenata di `HashMap`. Possiamo asserire brevemente che le tre implementazioni di riferimento di `Map`, ovvero `HashMap`, `TreeMap` e `LinkedHashMap`, hanno funzionalità e prestazioni equivalenti alle omologhe implementazioni di riferimento di `Set`: `HashSet`, `TreeSet` e `LinkedHashSet`.

Infine, l'interfaccia `ConcurrentMap` estende l'interfaccia `Map` e appartiene al package `java.util.concurrent`. Definisce metodi come `putIfAbsent()`, `remove()` e `replace()`. Tutte le azioni di inserimento e di rimozione degli elementi sono thread-safe e le sue implementazioni, come `ConcurrentHashMap`, sono adatte per sistemi dove c'è l'esigenza di avere mappe condivise da più thread che fanno operazioni su di esse.

16.7 Algoritmi e utilità

Le classi `Collections` ed `Arrays` offrono una serie di metodi statici che rappresentano utilità ed algoritmi utili con le collezioni.

In particolare la classe `Collections` offre particolari metodi (detti wrapper) per operare su collezioni in modo tale da modificare il loro stato. Tali metodi prendono in input una collezione e ne restituiscono un'altra con le proprietà cambiate. Per esempio i seguenti metodi:

- ❑ `public static Collection unmodifiable`
- ❑ `Collection(Collection c)`
- ❑ `public static Set unmodifiableSet(Set s)`
- ❑ `public static List unmodifiableList(List list)`
- ❑ `public static Map unmodifiableMap(Map m)`
- ❑ `public static SortedSet unmodifiableSortedSet(SortedSet s)`
- ❑ `public static SortedMap unmodifiableSortedMap(SortedMap m)`

restituiscono copie immutabili di ogni collezione. Infatti, provando a modificare la collezione, verrà sollevata una `UnsupportedOperationException`.

I seguenti metodi invece:

- ❑ `public static Collection synchroni`
- ❑ `zedCollection(Collection c)`
- ❑ `public static Set synchronizedSet(Set s)`
- ❑ `public static List synchronizedList(List list)`
- ❑ `public static Map synchronizedMap(Map m)`
- ❑ `public static SortedSet synchronizedSortedSet(SortedSet s)`
- ❑ `public static SortedMap synchronizedSortedMap(SortedMap m)`

sincronizzano tutte le tipologie di collezioni.

Per esempio, anche l'ArrayList si può sincronizzare sfruttando uno dei tanti metodi statici di utilità della classe Collections (da non confondere con Collection):

```
List<String> list = Collections.synchronizedList(new ArrayList<String>(...));
```

Abbiamo già asserito che in generale le prestazioni di un vector (cfr. appendice O online) sono inferiori rispetto a quelle di un ArrayList. La situazione in questo caso però si capovolge: il vector ha prestazioni superiori a quelle di un ArrayList sincronizzato.

Sincronizzare una collezione non significa sincronizzare anche il rispettivo Iterator. È quindi obbligatorio eseguire un ciclo con un Iterator quantomeno all'interno di un blocco sincronizzato come nel seguente esempio:

```
Collection<String> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    Iterator<String> i = c.iterator();
    while (i.hasNext())
        faQualcosa(i.next());
}
```

Questo perché l'iterazione avviene con chiamate multiple sulla collezione che vengono composte in un'unica operazione.

Il discorso è simile anche per le mappe ma è da approfondire direttamente sulla documentazione del metodo synchronizedMap().

Oltre ai metodi wrapper, Collections ha metodi che implementano complicati algoritmi per gli oggetti List come sort() (ordina con l'algoritmo merge sort), shuffle() (mischia gli elementi della lista specificata, il contrario di sort()), max() (restituisce l'elemento massimo), min() (restituisce l'elemento minimo), reverse() (inverte l'ordine degli elementi), binarySearch() (effettua la ricerca dicotomica), rotate() (ruota tutti gli elementi della lista specificata, del numero di posizioni specificato come secondo parametro), swap() (scambia gli elementi agli indici specificati), copy() (copia una lista in un'altra), e altri ancora.

Altri metodi (detti "di convenienza") permettono la creazione di collezioni immutabili di un numero definito di oggetti identici (metodo nCopies()) o di un oggetto singleton, che si può istanziare una sola volta (metodo singleton(), che restituisce un set facente da wrapper immutabile all'input specificato). Per esempio, potremmo rimuovere tutti gli elementi di un certo tipo da una collezione sfruttando il metodo removeAll() di Collection, e il metodo singleton() di Collections, in modo simile a:

```
collection.removeAll(Collections.singleton(oggettoDaRimuovere));
```

Per esempio, è possibile eliminare tutti gli oggetti null da una collezione:

```
collection.removeAll(Collections.singleton(null));
```

La classe `Arrays` contiene alcuni dei metodi-algoritmi di `Collections`, ma relativi ad array, come `sort()` e `binarySearch()`. Inoltre possiede il metodo `asList()`, che può trasformare un array in una `List` (precisamente un oggetto di tipo `ArrayList`, vedi documentazione). Per esempio:

```
List<String> l = Arrays.asList(new String[] { "1", "2", "3"});
```

crea un'implementazione di `List` a partire dall'array specificato al volo come parametro (di cui non rimarrà nessun puntamento e quindi verrà poi reso disponibile al Garbage Collector).

Con le versioni 5 e 6 di Java sono stati aggiunti nuovi metodi molto interessanti alla classe `Arrays`.

Per esempio, dalla versione 5 esistono nove versioni del metodo `toString(array)` soggetto a override per prendere in input tutte le possibili combinazioni di array di tipi primitivi e di `Object`. In questo modo si potranno stampare all'interno di parentesi quadre tutti gli elementi di un array (separati da virgole) senza ricorrere ad un ciclo. La versione 6 invece ha introdotto nove versioni del metodo `copyOfRange(array, from ,to)` per creare nuovi sotto-array a partire da array di dimensioni più grandi.

Infine la versione 8 ha introdotto il metodo `parallelSort()` che sfruttando l'algoritmo definito dal framework fork/join, ordina in maniera parallela un array, con prestazioni superiori rispetto a `sort()` nel caso di liste sufficientemente lunghe.

16.8 Introduzione alla libreria Stream API

La programmazione in generale si sta evolvendo per soddisfare le nuove architetture hardware dei nostri giorni. I programmati hanno sempre più spesso a che fare con ciò che viene definito “Big Data” (Terabyte di dati), con CPU multicore e cluster di computer per eseguire i propri programmi. Questo significa spesso usare programmazione parallela, ed è proprio sotto questo aspetto che Java era rimasta un po’ indietro. Linguaggi moderni come Groovy e Scala, e framework come il “MapReduce” di Google invece hanno puntato molto sulle nuove architetture. Ed è arrivato il momento quindi di aggiornare anche Java. La libreria “Stream API” è una libreria di alto livello che semplifica enormemente la programmazione Java (in particolare relativamente all’uso delle collezioni) e contemporaneamente offre supporto alla programmazione parallela. Tutto gira intorno alla definizione di `Stream`, una nuova interfaccia definita nel package `java.util.stream`. Uno `stream` rappresenta un flusso che ha come sorgente una collezione di dati, e su cui può eseguire operazioni complicate con uno sforzo di programmazione minimo, spesso basato su espressioni lambda e reference a metodi. Tipicamente uno `stream` viene creato a partire da una `collection`, tramite la chiamata al metodo `stream()`, ma ci sono tantissimi altri modi per ottenere oggetti `Stream`. Per esempio è possibile istanziare uno `Stream` passando un array al metodo statico `of()`:

```
Stream <String> stringsStream = Stream.of(arrayDiStringhe);
```

Ottiene passando all'altra versione del metodo `of()` direttamente dei valori:

```
Stream <String> stringsStream = Stream.of("Take", "The", "Time");
```

O ancora tramite il metodo statico `generate()` della stessa classe `Stream`. In questo caso il metodo `generate()` ci fornisce uno stream infinito perché ogni volta che vogliamo un valore dallo stream, ne verrà creato uno. Per esempio, il seguente frammento di codice istanzia uno `Stream` di infiniti numeri casuali, usando un reference al metodo `random()` della classe `Math` (cfr. paragrafo 11.1.8) che restituisce un numero casuale di tipo `double` maggiore o uguale di 0.0 e minore di 1:

```
Stream<Double> randomDoubles = Stream.generate(Math.random);
```

Soprattutto è possibile iterare sulle collezioni utilizzando un'oggetto `Stream`, che definisce anche un metodo `forEach()`, e da esso poi sfruttare le cosiddette “operazioni di aggregazione”. Queste ultime vengono utilizzate congiuntamente a espressioni lambda e reference a metodi, rendendo il codice estremamente potente e allo stesso tempo conciso. Queste espressioni infatti ritornano sempre un altro oggetto di tipo `Stream`, e di conseguenza è possibile concatenare le chiamate a queste funzioni. Queste concatenazioni di chiamate vengono definite comunemente **pipeline**. Per esempio, considerando l'oggetto `smartphones` dell'esempio precedente, la seguente istruzione:

```
smartphones.stream().filter(s->"Samsung".equals(s.getMarca())) .  
forEach(s->System.out.println(s));
```

stamperà:

```
Samsung Note 74
```

Infatti abbiamo chiamato il metodo `stream()` sulla collezione `smartphones`, ottenendo appunto un primo oggetto `Stream`. Su di esso abbiamo invocato il metodo `filter()` che prende in input un `Predicate` (cfr. paragrafo 15.3.1) che ricordiamo, era usato proprio per implementare dei test che ritornano un booleano. Quindi gli abbiamo passato una espressione lambda, che implementa un controllo con il metodo `equals()` sulla marca dell'elemento. Il metodo `filter()` ritorna nuovamente un'oggetto `Stream` relativo alla collezione filtrata che quindi conterrà un unico elemento (visto che nella collezione c'è un unico smartphone di marca Samsung). Su questo stream invochiamo il metodo `foreach()` della classe `Stream`, che esegue un'iterazione completa sugli elementi della collezione, e consente di eseguire un'operazione su ogni suo elemento. Anche in questo caso ci limitiamo a stampare gli elementi.

L'operazione `filter()` viene detta “operazione di aggregazione” perché progettata per restituire un nuovo stream, su cui invocare altre operazioni.

16.8.1 Definizioni di `Stream` e pipeline

Solitamente usiamo le collezioni per raggruppare elementi in un solo oggetto, per poi eseguire operazioni sui loro elementi all'interno di cicli. Uno **stream** (che in italiano possiamo tradurre come “flusso”) è una sequenza di elementi che si può iterare. L'iterazione è unica, una volta terminata, lo stream non è più utilizzabile. A differenza di una collezione, uno stream non immagazzina elementi,

ma permette di operare su questi elementi attraverso pipeline (che in italiano potremmo tradurre come “conduttrice” inteso come “tubo composto da diversi tubi per far passare gli stream”). Di fatto ha il grande vantaggio di consentire l’esecuzione di tali operazioni in maniera molto concisa. Il vantaggio più evidente è quando concateniamo le operazioni da fare sugli elementi della collezione tramite le pipeline. Possiamo definire **pipeline** come una sequenza di operazioni di aggregazione. Una pipeline è costituita da tre elementi:

1. **una sorgente** (in inglese **source**): potrebbe essere una collezione o un array. Ma esistono anche altri tipi di sorgente come canali di input/ output, oppure il metodo `generate()` della classe `Stream` stessa, e altro ancora. Una sorgente dovrebbe mettere a disposizione un metodo per ottenere un oggetto `Stream`.
2. Zero o più **operazioni di aggregazione** (o **intermediate**): queste operazioni hanno la caratteristica di ritornare un nuovo oggetto `Stream`. Nell’esempio precedente abbiamo visto come operazione di aggregazione il metodo `filter()`. Altri tipici metodi intermedi sono i metodi `map()`, `mapToDouble()` etc. Tutti i metodi intermedi vengono chiamati solo all’ultimo momento (si dice che sono metodi “lazy”, in italiano “pigri”), nel senso che saranno invocati solo quando sarà invocato il metodo terminale, che ovviamente ha bisogno dei metodi intermedi per poter essere eseguito.
3. Una **operazione terminale**: un metodo che restituisce un risultato che non è un oggetto `Stream`. Nell’esempio precedente abbiamo visto come operazione terminale il metodo `forEach()`.

In modo astratto la terminologia inglese parla metaforicamente di flussi e tubi. In pratica gli `stream` (i flussi) non immagazzinano dati ma trasportano gli elementi dalla sorgente (nel nostro caso una collezione) attraverso la tubatura (la pipeline), trasformando (filtrando, modificando etc.) il risultato finale. Questo tipo di astrazione è simile a quella che da sempre viene proposta per la libreria dell’Input-Output tramite `InputStream` e `OutputStream` come vedremo nel prossimo modulo.

Per esempio, sempre considerando la classe `Smartphone` di prima, e supponendo che il metodo `getSmartphones()` ritorni una collezione di oggetti `Smartphone`, allora potremmo calcolare la media del prezzo degli smartphone di marca Samsung con il seguente codice:

```
Collection<Smartphone> smartphones = getSmartphones();  
double average = smartphones.stream()  
    .filter(s -> s.getMarca().equals("Samsung"))  
    .mapToDouble(Smartphone::getPrezzo).average().getAsDouble();
```

Si noti che la pipeline specificata concatena ben cinque metodi. Dalla sorgente `smartphones` otteniamo prima uno `Stream` invocando il metodo `stream()`. Su questo `stream` andiamo a chiamare il metodo `filter()` a cui passiamo un’espressione lambda per filtrare tutti i modelli di marca Samsung. Questo metodo restituisce a sua volta uno `Stream` su cui chiamiamo il metodo `mapToDouble()` a cui passiamo un reference al metodo `getPrezzo()` della classe `Smartphone`. Il metodo `mapToDouble()` restituirà uno `stream` (in questo caso di tipo `DoubleStream` ovvero uno

stream che contiene solo valori `Double`) su una collezione di oggetti `Double` con i prezzi degli smartphone Samsung filtrati. Su quest'ultimo stream chiamiamo il metodo `average()`, che esegue un calcolo matematico degli elementi della collezione di `Double` e ritorna un oggetto di tipo `OptionalDouble`. Infine invochiamo il metodo `getAsDouble()` che ritorna il double encapsulato nell'oggetto `OptionalDouble`.

Un metodo come `average()` viene detto “reduction operation” (in italiano “operazione di riduzione”), in quanto ritorna un unico valore calcolato da più elementi dello stream.

16.8.2 Classi Optional

Si noti che `OptionalDouble` è una delle importanti classi di Java dette “classi optional”. Le classi optional sono delle classi wrapper che consentono di evitare `NullPointerException`. Queste infatti possono immagazzinare valori di un certo tipo o `null`. Esistono le classi `OptionalDouble`, `OptionalInt`, `OptionalLong` e `Optional<T>`. Tuttavia bisogna utilizzare queste classi nella maniera corretta per non incorrere in problemi. Infatti, è vero che non lanciano `NullPointerException`, ma se nell'esempio precedente il metodo `getSmartphones()` non avesse restituito smartphone di marca Samsung, avremmo avuto il seguente output:

```
Exception in thread "main" java.util.NoSuchElementException:  
No value present  
at java.util.OptionalDouble.getAsDouble(OptionalDouble.java:118)  
at TestPipeline.main(TestPipeline.java:11)
```

Quindi qual è il vantaggio di avere una `NoSuchElementException` in luogo di una `NullPointerException`? Le classi optional hanno dei metodi a cui è possibile passare espressioni lambda o valori di ritorno alternativi. Ci limiteremo ad un unico esempio sebbene le classi optional hanno diversi metodi interessanti da esplorare. Il metodo `orElse()`, permette di specificare un elemento alternativo nel caso l'oggetto `optional` sia vuoto:

```
Optional<Smartphone> found = smartphones.stream()  
.filter(s -> s.getMarca().equals("Samsung")).findFirst();  
System.out.println(found.orElse(new Smartphone("Samsung", "Note 3")));
```

Nell'esempio abbiamo usato il metodo `findFirst()`, che è un metodo terminale che torna un oggetto `Optional` con il primo elemento dello stream, oppure un `Optional` vuoto. Prima di stampare invochiamo su di esso il metodo `orElse()` a cui passiamo un valore alternativo. Questo verrà stampato nel caso l'`Optional` sia vuoto. Abbiamo evitato così i soliti verbosi controlli di nullità.

16.8.3 Reduction Operations

Abbiamo già asserito che il metodo `average()` è considerato una **reduction operation** (in italiano una **operazione di riduzione**) ovvero un metodo che a partire dagli elementi di uno stream, ritorna un

solo valore. Altre tipiche implementazioni di operazioni di riduzione sono i metodi `max()`, `min()`, `sum()` e `count()` che hanno delle implementazioni facilmente immaginabili. Per esempio, con la seguente istruzione:

```
long count = smartphones.stream()
    .filter(s -> s.getMarca().equals("Apple")).count();
```

contiamo il numero di smartphone di marca Apple.

Esistono però anche altre operazioni di riduzione che ritornano collezioni invece che singoli elementi. Stiamo parlando dei metodi `reduce()` e `collect()`.

16.8.3.1 Metodo `reduce()`

Il metodo `reduce()` esiste in più versioni (overload). La più famosa prende in input due parametri:

1. un oggetto detto **identity** che rappresenta sia il valore iniziale che il valore da restituire nel caso non ci siano elementi nello stream;
2. e un oggetto di tipo `BinaryOperator` chiamato **accumulator**. `BinaryOperator` è un'altra interfaccia funzionale che estende `BiFunction` (cfr. paragrafo 15.3.4) e quindi stiamo parlando di passare una espressione lambda con due parametri.

Per esempio, per ritornare il prezzo totale degli smartphone nella collezione di una certa marca, potremmo scrivere:

```
long prezzoTotale = smartphones.stream()
    .filter(s -> s.getMarca().equals("Samsung"))
    .map(Smartphone::getPrezzo).reduce(0, (x,y) -> x+y);
```

anche se nel caso specifico sarebbe stato più semplice usare il metodo `sum()`.

Il metodo `reduce()` però, calcola ogni volta un valore, e questo, nelle situazioni più complesse della precedente, potrebbe rappresentare un problema per le prestazioni. Per casi come questo esiste il metodo `collect()`.

16.8.3.2 Metodo `collect()` e la classe `Collectors`

Questo metodo viene utilizzato spesso congiuntamente ai metodi statici messi a disposizione della classe `Collectors`. Anche in questo caso infatti il metodo possiede diversi overload, ma la versione più famosa è quella che prende in input un oggetto `Collectors`. Questa classe contiene metodi come `toList()`, `toCollection()`, `reducing()`, `joining()` e `groupingBy()`, di semplice utilizzo ed interpretazione. Per esempio con la seguente istruzione:

```
List<String> list1 = smartphones.stream().map(Smartphone::getModello)
    .collect(Collectors.toList());
```

otteniamo una lista di stringhe di tutti i modelli della collezione di Smartphone. Invece con la seguente istruzione:

```
Set<String> list2 = smartphones.stream().map(Smartphone::getMarca)
    .collect(Collectors.toCollection(TreeSet::new));
```

otteniamo un Set (quindi senza duplicati) di stringhe contenente le marche degli smartphone. Si noti che in questo caso abbiamo dovuto passare un reference a metodo di TreeSet, per specificare quale collezione utilizzare.

Sfruttando la nuova classe StringJoiner di java.util (che permette di associare un separatore a varie stringhe), il metodo joining() di Collectors ci permette di stampare la lista dei modelli separati da virgole senza più il bisogno di creare algoritmi per capire se bisogna aggiungere o meno il separatore, il codice è banale:

```
String modelliConSeparatore = smartphones.stream()
    .map(Smartphone::toString).collect(Collectors.joining(", "));
```

e stamperà:

```
HTC One M8; Samsung Note 3; Apple Iphone 5S;
```

Interessante è raggruppare gli elementi delle collezioni per marca:

```
Map<String, List<Smartphone>> map = smartphones.stream()
    .collect(Collectors.groupingBy(Smartphone::getMarca));
```

Si noti come viene restituita una mappa che associa alla marca gli smartphone... uno sforzo davvero minimo rispetto a quello che avremmo dovuto scrivere prima di Java 8!

Un'altro interessante utilizzo di Collectors sono i metodi “summarizing”, che consentono di ottenere un oggetto di tipo DoubleSummaryStatistics:

```
DoubleSummaryStatistics stats = smartphones.stream()
    .collect(Collectors.summarizingDouble(Smartphone::getPrezzo));
```

Quest'oggetto contiene tutte le informazioni ottenibili dalle operazioni di riduzione basilari. Infatti stampando il suo metodo toString(), otterremo il seguente output:

```
DoubleSummaryStatistics{count=8, sum=3326,000000, min=70,000000,
average=415,750000, max=721,000000}
```

16.8.4 Stream paralleli e prestazioni

In un esempio precedente filtravamo e stampavamo gli elementi Smartphone usando una pipeline che usava uno Stream:

```
smartphones.stream().filter(s->"Samsung".equals(s.getMarca()))
    .forEach(s->System.out.println(s));
```

Allo stesso modo avremmo potuto anche usare uno `Stream` che eseguisse le stesse operazioni in maniera parallela. Questo si ottiene semplicemente chiamando il metodo `parallelStream()` sulla collezione, in luogo del metodo `stream()`. Per il resto non cambia nulla:

```
smartphones.parallelStream().filter(s->"Samsung".equals(s.getMarca()))
    .forEach(s->System.out.println(s));
```

Questo “stream parallelo”, per filtrare gli elementi della collezione, utilizza un algoritmo di tipo fork/join. Ovvero divide la collezione in più parti e assegna l’operazione di filtraggio di ogni parte a un thread diverso. Quando tutti i thread hanno terminato il loro processo, i risultati vengono ricongiunti per ottenere lo stesso risultato che si sarebbe ottenuto filtrando tutta la collezione interamente. Il vantaggio è che i vari thread potrebbero svolgere il lavoro in parallelo se ci sono abbastanza core di CPU a disposizione sulla macchina, riducendo il tempo di esecuzione del processo. L’esempio è puramente didattico in quanto è completamente inutile usare uno stream parallelo su una collezione di soli tre elementi. Inoltre questo tipo di esecuzione parallela ha un costo: quello di dover ogni volta suddividere la collezione in più collezioni per poi effettuare il ricongiungimento. Quindi non bisogna aspettarsi che i tempi di esecuzione vengano ridotti drasticamente usando uno stream parallelo, nella maggior parte dei casi. Il vantaggio in termini di prestazioni è da calcolare rispetto allo stessa pipeline eseguita su uno stream non parallelo, e non alle stesse operazioni eseguite con cicli `foreach` o iteratori. Per esempio, supponiamo di voler contare quante volte una certa parola viene usata all’interno de “La Divina Commedia”. Possiamo creare un programmino che compara le varie prestazioni per eseguire questo processo. Riportiamo qui sotto solo la parte relativa agli stream:

```
count = words.stream()
    .filter(word -> word.equals(wordToSearch)).count();
// ...
count = words.parallelStream()
    .filter(word -> word.equals(wordToSearch)).count();
// ...
```

Ma il programma può effettuare la stessa operazione anche eseguendo un ordinario ciclo `foreach`, usando un `Iterator` e un metodo `forEach()` direttamente sulla collezione (è possibile visionare il listato completo tra gli esempi on line). Tra un’esecuzione e un’altra registriamo i millisecondi impiegati per eseguire il processo. Il risultato finale (che cambia ad ogni esecuzione) è:

Tempo ciclo foreach	= 7 millisecondi	count = 517
Tempo oggetto Iterator	= 2 millisecondi	count = 517
Tempo metodo forEach()	= 41 millisecondi	count = 517
Tempo stream ordinario	= 10 millisecondi	count = 517
Tempo stream parallelo	= 51 millisecondi	count = 517

Quindi come si può vedere gli stream non offrono in generale le prestazioni migliori, questo dipende dal tipo di operazioni che vengono eseguite. Dando per scontato di avere sul proprio processore i necessari core per effettuare calcoli paralleli, l'effettivo vantaggio in termini di prestazioni si ottiene quando abbiamo situazioni in cui processando un certo elemento ci sia la possibilità di blocco. Tale blocco potrebbe essere determinato da un eventuale attesa da parte del processo di input che arriva da fonti esterne rispetto alla nostra macchina, come una risorsa di rete o un database remoto (Input/Output e database sono gli argomenti dei prossimi due moduli). In tal caso uno stream parallelo potrebbe ridurre drasticamente i tempi di processo totali. Infatti, se nel processare un certo elemento si verificasse questa attesa, solo il parallelismo ci permetterebbe di continuare la nostra esecuzione sugli altri elementi. In tutti gli altri casi invece il ciclo aspetterà che l'esecuzione del processo sull'elemento bloccante termini sprecando tempo prezioso.

Quello che bisogna evitare è usare il parallelismo senza che ce ne sia bisogno. Il fatto che basti invocare il metodo `parallelStream()` per implementare funzionalità di parallelismo, potrebbe risultare troppo invitante per alcuni programmatore poco esperti. In realtà si può incorrere facilmente in errori difficili da risolvere. Ciononostante, problemi di parallelismo possono accadere anche senza usare stream paralleli. Per esempio il seguente frammento di codice lancia una `java.util.ConcurrentModificationException`:

```
List<String> stringhe =  
    new ArrayList<>(Arrays.asList("stringa1", "stringa2"));  
String stringaConcatenata = stringhe.stream()  
    .peek(s -> stringhe.add("stringa3")).collect(Collectors.joining(", "));
```

infatti si sta provando ad aggiungere “al volo” un elemento alla collezione durante l'esecuzione della pipeline, col metodo `peek()`. Ma i metodi intermedi sono lazy, e il metodo `peek()` inizia la sua esecuzione solo dopo che è partita l'esecuzione del metodo `reduce()`.

Riepilogo

La conoscenza approfondita del framework Collections è fondamentale per programmare in Java, e con l'introduzione della libreria Stream API il modo in cui scriviamo i nostri programmi sarà rivoluzionato. In questo modulo abbiamo dapprima visto come la gerarchia delle collezioni parta dall'interfaccia `Collection` che definisce decine di metodi fondamentali. Abbiamo dedicato particolare attenzione a come poter iterare sulle collezioni (ciclo `foreach`, `Iterator` e metodo `forEach()`). Poi abbiamo mostrato l'interfaccia `List` e le sue implementazioni fondamentali come `ArrayList` e `LinkedList`. Siamo passati quindi ad esplorare l'interfaccia e le implementazioni di `Set` e `SortedSet` come `TreeSet`, `HashSet` e `LinkedHashSet`. Successivamente abbiamo affrontato l'argomento delle code con le interfacce `Queue`, `Deque` le loro implementazioni e la loro controparte concorrente (`BlockingQueue`). In seguito abbiamo esplorato il mondo delle mappe con le interfacce `Map`, `SortedMap` e implementazioni più importanti. Infine abbiamo studiato una serie di metodi offerti in particolar modo dalle classi `Collections` ed `Arrays`.

Nella seconda parte del modulo abbiamo introdotto la libreria **Stream API** che rivoluzionerà il nostro modo di programmare. Abbiamo dapprima dato le definizioni di `stream` e `pipeline`, ed abbiamo presentato diversi esempi. In particolare ci siamo soffermati sui metodi `filter()`,

`reduce()` e `collect()`. In ultimo abbiamo anche introdotto le **classi optional** e la **programmazione parallela** con gli stream.

Esercizi modulo 16

Esercizio 16.a) Framework Collections, Vero o Falso:

1. `Collection`, `Map`, `SortedMap`, `Set`, `List` e `SortedSet` sono interfacce e non possono essere istanziate.
2. Un `Set` è una collezione ordinata di oggetti; una `List` non ammette elementi duplicati ed è ordinata.
3. Le mappe non possono contenere chiavi duplicate ed ogni chiave può essere associata ad un solo valore.
4. Esistono diverse implementazioni astratte da personalizzare nel framework come `AbstractMap`.
5. Una `HashMap` è più performante rispetto ad una `Hashtable` perché non è sincronizzata.
6. Una `HashMap` è più performante rispetto ad un `TreeMap` ma quest'ultima, essendo un'implementazione di `SortedMap`, gestisce l'ordinamento.
7. `HashSet` è più performante rispetto a `TreeSet` ma non gestisce l'ordinamento.
8. `Iterator` ed `Enumeration` hanno lo stesso ruolo ma quest'ultima permette durante le iterazioni di rimuovere anche elementi.
9. `ArrayList` ha prestazioni migliori rispetto a `Vector` perché non è sincronizzata, ma entrambe hanno meccanismi per ottimizzare le prestazioni.
10. La classe `Collections` è un lista di `Collection`.

Esercizio 16.b) Stream API, Vero o Falso:

1. `Stream` è una classe che implementa `Collection`.
2. È possibile iterare su uno stream in entrambe le direzioni.
3. Un pipeline è costituita da una sorgente, metodi di aggregazione opzionali e un metodo terminale.
4. Il metodo `map()` è da considerarsi un metodo di aggregazione.
5. `Optional` è un'interfaccia che permette di evitare di avere a che fare con le `NullPointerException`.
6. I metodi di riduzione sono operazioni di aggregazione.

- Il metodo `joining()` di Stream permette di concatenare stringhe con dei separatori di tipo stringa.
- La classe `DoubleSummaryStatistics` è un particolare tipo di stream.
- Il metodo `parallelStream()` ritorna uno stream capace di usare l'algoritmo Fork/Join per eseguire operazioni sugli elementi di una collezione.
- Uno stream è istanziabile solo a partire da una collezione.

Soluzioni esercizi modulo 16

Esercizio 16.a) Framework Collections, Vero o Falso:

- Vero.**
- Falso.**
- Vero.**
- Vero.**
- Vero.**
- Vero.**
- Vero.**
- Falso.**
- Vero.**
- Falso.**

Esercizio 16.b) Stream API, Vero o Falso:

- Falso**, Stream è un'interfaccia.
- Falso.**
- Vero.**
- Vero.**
- Falso**, Optional è una classe (peraltro dichiarata `final` e quindi non estendibile).
- Falso**, sono operazioni terminali.
- Falso**, il metodo `joining()` appartiene alla classe `Collectors`.
- Falso.**
- Vero.**
- Falso.**

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi

<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Comprendere l'utilità del framework Collections (unità 16.1)	<input type="checkbox"/>	
Conoscere le principali interfacce del framework Collections ed i metodi che espongono (unità 16.2, 16.3, 16.4, 16.5, 16.6)	<input type="checkbox"/>	
Conoscere le principali implementazioni del framework Collections ed i metodi che espongono (unità 16.2, 16.3, 16.4, 16.5, 16.6)	<input type="checkbox"/>	
Saper utilizzare le classi d'utilità Arrays e Collections ed i metodi che espongono (unità 16.7)	<input type="checkbox"/>	
Comprendere l'utilità della libreria Stream API (unità 16.8)	<input type="checkbox"/>	
Saper utilizzare stream e pipeline (unità 16.8)	<input type="checkbox"/>	
Saper utilizzare metodi di aggregazione, di riduzione e di parallelismo (unità 16.8)	<input type="checkbox"/>	

Note:

Input-Output

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Aver compreso il pattern Decorator (unità 17.1, 17.2).
- ✓ Saper riconoscere nelle classi del package `java.io` i ruoli definiti nel pattern Decorator (unità 17.3).
- ✓ Capire le fondamentali gerarchie del package `java.io` (unità 17.3).
- ✓ Avere confidenza con i tipici problemi che si incontrano con l'input-output, come la serializzazione degli oggetti e la gestione dei file (unità 17.4).
- ✓ Avere confidenza con il nuovo modello per l'input output di Java 7 denominato NIO 2.0 (unità 17.5).

In questo modulo affronteremo finalmente la libreria sull'Input/Output. È una libreria complessa e vastissima, ciononostante intuitiva e potente. Anche in questo caso i concetti fondamentali si basano su `OutputStream` ed `InputStream`, da non confondere con gli oggetti di tipo `stream`. Negli anni tantissimi altre classi e metodi sono stati introdotti e particolarmente importante è la libreria nota come NIO 2, che ha aggiunto altre funzionalità di alto livello per lavorare con i file in modo più semplice.

17.1 Introduzione all'input-output

Spesso le applicazioni hanno bisogno di utilizzare informazioni lette da fonti esterne, o inviare informazioni a destinazioni esterne. Per informazioni intendiamo non solo stringhe, ma anche oggetti, immagini, suoni etc. Per fonti o destinazioni esterne all'applicazione invece intendiamo file, dischi, reti, memorie o altri programmi. In questo modulo vedremo come Java consenta di gestire la lettura (input) da fonti esterne e la scrittura su destinazioni esterne (output). In particolare introduciamo il package `java.io`, croce e delizia dei programmati Java. Il package in questione è molto vasto e anche abbastanza complesso. Conoscerne ogni singola classe è un'impresa ardua e soprattutto inutile. Per poter gestire l'input-output in Java conviene piuttosto capirne la filosofia che ne è alla base, regolata dal design pattern noto come Decorator (cfr. appendice D per la definizione di pattern). Non comprendere il pattern Decorator implicherà fare sempre fatica nel districarsi tra le classi di `java.io`. Al lettore quindi si raccomanda la massima concentrazione, visto che anche il pattern stesso è abbastanza complesso.

17.2 Pattern Decorator

È facile riconoscere nella gerarchia delle classi di `java.io` il modello di classi definito dal pattern Decorator. Si tratta di un pattern GoF strutturale (cfr. appendice D) decisamente complesso ma incredibilmente potente.

Nelle prossime righe viene proposto un esempio del pattern Decorator per poter dare un'idea al lettore della sua utilità. È però possibile (ma non consigliabile) passare direttamente a leggere la descrizione del package, se l'argomento pattern non interessa o lo si ritiene troppo complesso.

17.2.1 Descrizione del pattern

Il pattern Decorator permette al programmatore di implementare, tramite una particolare gerarchia di ruoli, una relazione tra classi che rappresenta un'alternativa dinamica alla statica ereditarietà. Sarà possibile aggiungere responsabilità addizionali agli oggetti al runtime, piuttosto che creare una sottoclasse per ogni nuova responsabilità.

Sarà anche possibile ovviare a questo tipo di problema: supponiamo di voler aggiungere ad una certa classe, `ClasseBase`, delle responsabilità (chiamiamole `r1`, `r2`, `r3`, e `r4`) di pari dignità, ovvero ognuna indipendente dalle altre.

Potrebbe però servire anche creare classi che hanno più di una di queste responsabilità, per esempio `r1`, `r2` e `r4`.

In questo caso particolare l'ereditarietà da sola non riesce a soddisfare i nostri bisogni. Partendo dalla gerarchia in Figura 17.1 il lettore può provare a cercare una soluzione al problema. Probabilmente ci sarà un numero di classi pari a 2 elevato al numero delle varie responsabilità, nel nostro esempio 2 elevato alla quarta potenza, ovvero 16. Inoltre, causa l'impossibilità di implementare l'ereditarietà multipla con le classi, non sarà possibile ottenere un risultato accettabile dal punto di vista object oriented.

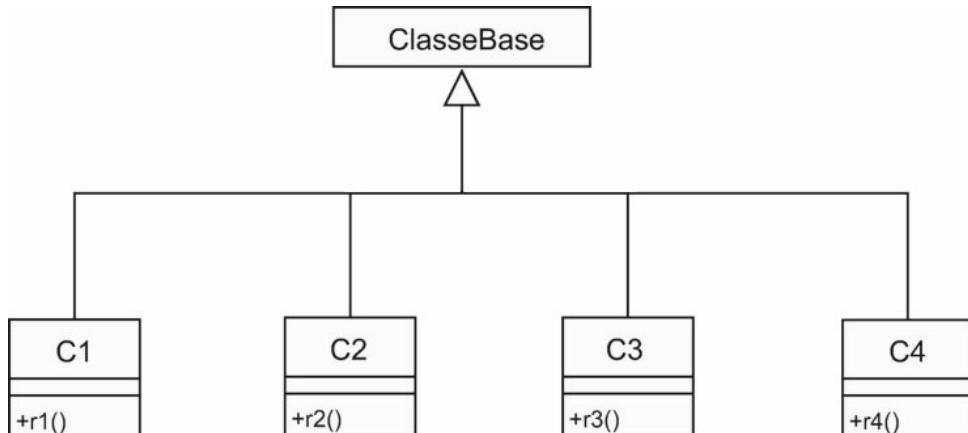


Figura 17.1 - Ereditarietà.

Consigliamo al lettore però di non impegnarsi troppo per trovare la soluzione a questo problema, perché ne esiste una geniale già a disposizione di tutti.

Nel pattern Decorator esistono i seguenti ruoli:

- ❑ **Component**: un'interfaccia (o classe astratta) che definisce una o più operazioni astratte da implementare nelle sottoclassi.
- ❑ **ConcreteComponent**: questo ruolo è interpretato da una o più classi non astratte, che implementano Component.
- ❑ **Decorator**: un'altra estensione di Component che potrebbe anche essere dichiarata astratta. Questa deve semplicemente obbligare le sue sottoclassi (**ConcreteDecorator**) non solo ad implementare Component, ma anche a referenziare un Component con un'aggregazione (cfr. appendice L per la definizione di aggregazione) (vedi Fig. 17.2). Questo ruolo può essere considerato opzionale.
- ❑ **ConcreteDecorator**: implementazione di Decorator, che come abbiamo già asserito dovrà implementare Component e le sue operazioni e mantenere un riferimento verso un Component.

Tutto ciò viene riassunto con il diagramma in Figura 17.2.

Facciamo un esempio per comprendere come funziona questo pattern.

Come al solito è necessario calarsi in un contesto con un po' di fantasia e flessibilità. Supponiamo di voler realizzare un'applicazione grafica che consenta di aggiungere effetti speciali (come un effetto 3D e un effetto trasparente) alle nostre immagini. Per realizzare ciò occorre identificare la classe Immagine come ConcreteComponent e le classi Effetto3DDecorator e TrasparenteDecorator come ConcreteDecorator, come mostrato in Figura 17.3.

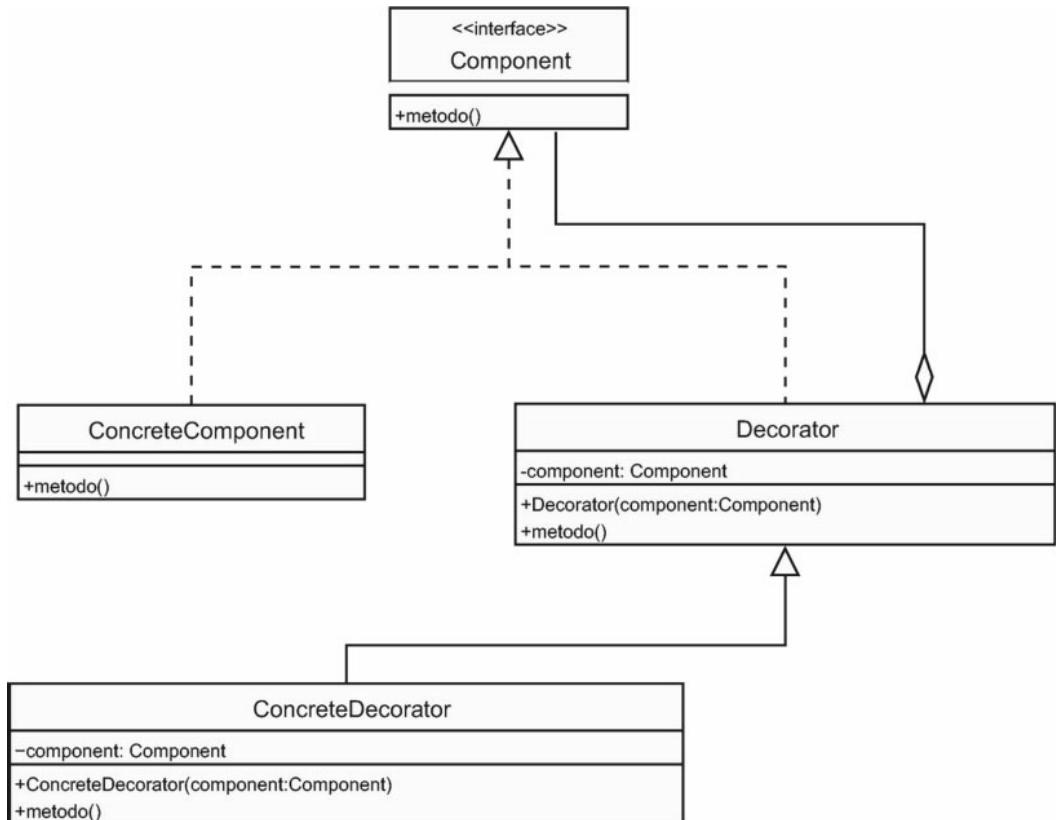


Figura 17.2 - Modello del pattern Decorator.

A questo punto analizziamo il seguente codice:

```

1 Immagine monnaLisa = new Immagine();
2 monnaLisa.visualizza();
3 Effetto3DDecorator monnaLisa3D = new Effetto3DDecorator(monnaLisa);
4 monnaLisa3D.visualizza();
5 TrasparenteDecorator monnaLisa3DTrasparente = new
    TrasparenteDecorator(monnaLisa3D);
6 monnaLisa3DTrasparente.visualizza();

```

Alle righe 1 e 2 viene istanziata e visualizzata l'immagine `monnaLisa`. Alla riga 3 entra in scena il primo decorator che viene istanziato aggregandogli l'oggetto `monnaLisa` appena istanziato. Questo è il punto più complesso e ha bisogno di essere ben analizzato. Per prima cosa notiamo che ogni decorator ha un unico costruttore che prende in input obbligatoriamente (per il concetto di aggregazione cfr. appendice L) un `Component`. Ovvero non può esistere un oggetto `Decorator` senza che aggreghi un `Component` (in questo caso `Immagine` ha ruolo di `ConcreteComponent`). Di fatto un effetto speciale non può esistere senza l'oggetto da decorare. È questo il punto chiave per

comprendere il pattern.

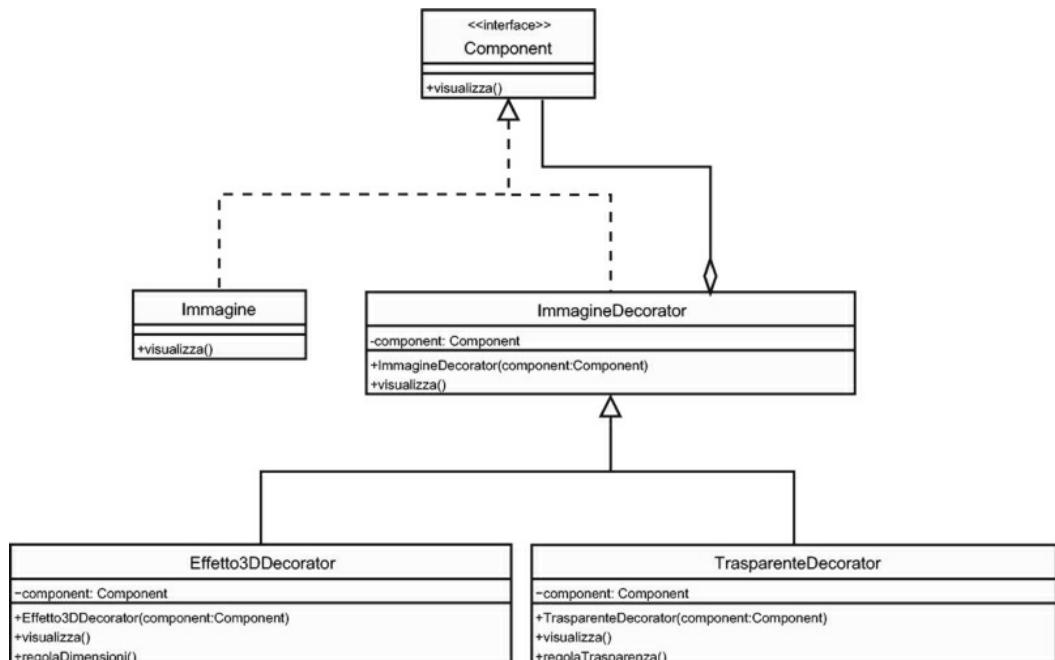


Figura 17.3 - Esempio di implementazione di Decorator.

Alla riga 4 viene invocato il metodo `visualizza()` direttamente sull'oggetto `Effetto3DDecorator` (`monnaLisa3D`) che contiene l'immagine. Notiamo come l'oggetto `monnaLisa3D` (di tipo `Effetto3DDecorator`) se da un lato rappresenta solo una decorazione dell'immagine, in questo pattern diventa proprio l'oggetto immagine decorato. Ecco perché abbiamo preferito chiamarlo `monnaLisa3D` nonostante sia di tipo `Effetto3DDecorator`. Sarebbe più naturale che un oggetto di tipo `Immagine` (o una sua sottoclasse) rimanesse l'oggetto contenitore, ma bisogna abituarsi all'idea per ottenere i benefici di questo pattern.

Notiamo inoltre che il metodo `visualizza()` di un `Decorator` farà sicuramente uso anche del metodo `visualizza()` dell'`Immagine`. Per esempio il metodo `visualizza()` della classe `Effetto3DDecorator` potrebbe essere codificato più o meno nel modo seguente:

```
public void visualizza() {
    //codice per generare l'effetto 3D . . .
    component.visualizza();
}
```

Il riferimento `component` punta all'oggetto `monnaLisa` di tipo `Immagine`.

Alla riga 5 si ripete un'altra “decorazione”, ma stavolta il decoratore (`monnaLisa3DTrasparente`

di tipo `TrasparenteDecorator`) invece di decorare un oggetto di tipo `Immagine`, agisce su un altro decoratore (`monnaLisa3D` di tipo `Effetto3DDecorator`) che ha già operato su un oggetto di tipo `Immagine` (`monnaLisa`). Ciò è possibile grazie al fatto che ogni `Decorator` deve decorare per forza un `Component` e `Component` è implementato anche dai `Decorator`.

Alla riga 6 viene invocato il metodo `visualizza()` direttamente sull'oggetto `TrasparenteDecorator` (`monnaLisa3DTrasparente`) che contiene il decoratore contenente a sua volta l'immagine.

Anche in questo caso il metodo `visualizza()` di `TrasparenteDecorator` farà sicuramente uso anche del metodo `visualizza()` del suo `component` che aggrega. Stavolta però il `component` aggregato non è una semplice immagine ma un oggetto di tipo `Effetto3DDecorator` (`monnaLisa3D`) che a sua volta aggregava l'oggetto `monnaLisa` di tipo `Immagine`.

Concludendo, il pattern è sicuramente complesso e questo non gioca a suo favore, ma è incredibilmente potente. Basti pensare che, facendo riferimento all'esempio appena presentato, ci sono almeno i seguenti vantaggi:

1. il numero delle classi da creare è molto minore che in qualsiasi soluzione basata sulla ereditarietà. Per esempio non deve esistere la classe `Immagine3DTrasparente`.
2. Se nascono nuovi effetti speciali, basterà aggiungerli come `decorator` senza che tutto il resto del codice ne risenta.
3. Al runtime si può creare senza sforzi eccessivi qualsiasi combinazione basata sui `decorator`.

17.3 Descrizione del package

L'implementazione del pattern `Decorator` per la gestione dell'input-output in Java si è sicuramente rivelata la soluzione ideale. Infatti, come già asserito nell'introduzione, tale package deve mettere a disposizione dell'utente classi per realizzare un qualsiasi tipo di lettura (input) e un qualsiasi tipo di scrittura (output). Le fonti di lettura e le destinazioni di scrittura sono molte e in futuro potrebbero nascerne di nuove. Il pattern `Decorator` permette quindi di realizzare qualsiasi tipo di comunicazione con fonti di destinazioni esterne usando un limitato numero di classi. Nonostante ciò, il numero di classi del package `java.io` rimane comunque alto. Fortunatamente non bisogna conoscerle tutte; la documentazione ufficiale serve proprio a questo e, conoscendo il pattern `Decorator`, potremo riconoscere i ruoli di ogni classe.

Partiamo dal concetto fondamentale che è alla base del discorso: lo **stream** (in italiano **flusso**).

Per prelevare informazioni da una fonte esterna (un file, una rete etc.), un programma deve aprire uno stream su essa e leggerne le informazioni in maniera sequenziale. La Figura 17.4 mostra graficamente l'idea.

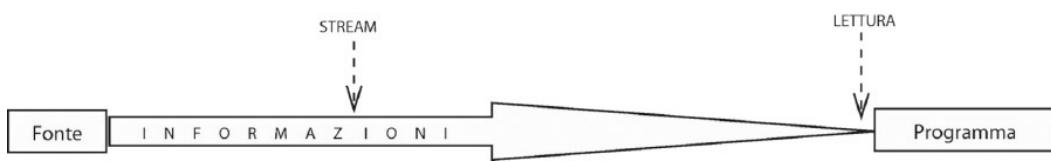


Figura 17.4 - Rappresentazione grafica di un input.

Allo stesso modo un programma può inviare ad una destinazione esterna aprendo uno stream su essa e scrivendo le informazioni sequenzialmente, come mostrato in Figura 17.5.

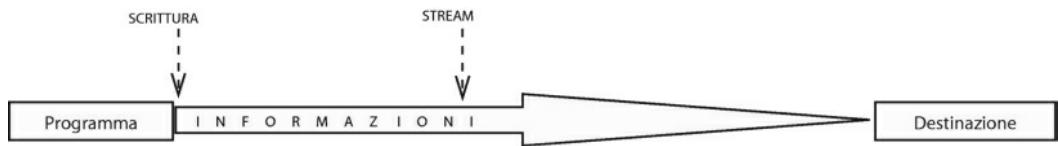


Figura 17.5 - Rappresentazione grafica di un output.

In altre parole, aprire uno stream, da una fonte o verso una destinazione, significa aprire verso questi punti terminali un canale di comunicazione, dove far passare le informazioni. Con `java.io` non importa di che tipo siano le informazioni, né con che fonti o destinazioni abbiamo a che fare. Gli algoritmi per scrivere o leggere, infatti, sono sostanzialmente sempre gli stessi.

Per l'input bisogna:

1. aprire lo stream;
2. leggere tutte le informazioni dallo stream fino a quando non terminano;
3. chiudere lo stream.

Per l'output bisogna:

1. aprire lo stream;
2. scrivere tutte le informazioni tramite lo stream fino a quando non terminano;
3. chiudere lo stream.

Il package `java.io` contiene una collezione di classi che supportano tali algoritmi per leggere e scrivere. Le classi di tipo stream sono divise in due gerarchie separate (anche se simili) in base al tipo di informazione che devono trasportare (byte o caratteri).

17.3.1 I Character Stream

`Reader` e `Writer` sono le due superclassi astratte per i “character stream” (“flussi di caratteri”) in `java.io`. Queste due classi hanno la caratteristica di obbligare le sottoclassi a leggere e scrivere dividendo i dati in “pezzi” di 16 bit ognuno, quindi compatibili con il tipo `char` di Java. Le sottoclassi di `Reader` e `Writer` hanno i ruoli del pattern Decorator. Il ruolo di `Component` è interpretato proprio da `Reader` (`eWriter`). Le sottoclassi di `Reader` (`eWriter`) implementano stream speciali. Alcune di queste hanno il ruolo di `ConcreteComponent` (dovremmo dire `ConcreteReader` e `ConcreteWriter`) e da sole possono attaccarsi ad una fonte (o ad una destinazione) e leggere (o scrivere) subito, quantomeno con un algoritmo sequenziale implementato nei metodi `read()` (o `write()`) ereditati da `Reader` (o `Writer`). Questi stream vengono anche detti

“Node Stream” (in italiano “flussi nodo”).

Altre sottoclassi di Reader (e Writer) invece interpretano il ruolo di ConcreteDecorator. Tali stream sono anche detti “Processing Stream” e senza aggregazioni (cfr. appendice L) a ConcreteReader (o ConcreteWriter) non si possono neanche istanziare. Il più delle volte lo scopo dei decoratori di stream è quello di migliorare le prestazioni o facilitare la lettura (o la scrittura) delle informazioni negli stream, fornendo metodi adeguati allo scopo, ed evitando così cicli noiosi e poco eleganti. Nelle Figure 17.6 e 17.7 sono mostrate le gerarchie delle classi principali per quanto riguarda rispettivamente Reader e Writer, con i decoratori colorati in grigio.

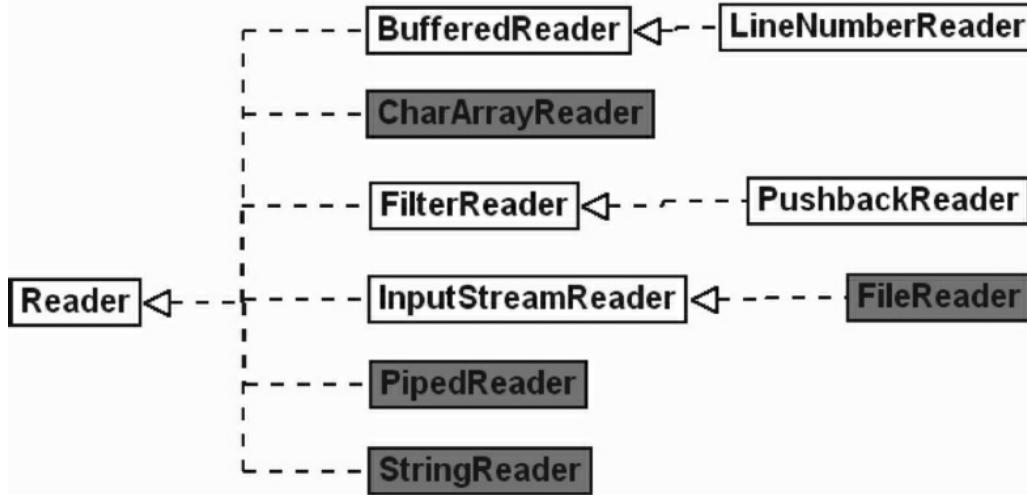


Figura 17.6 - Gerarchia dei Reader.

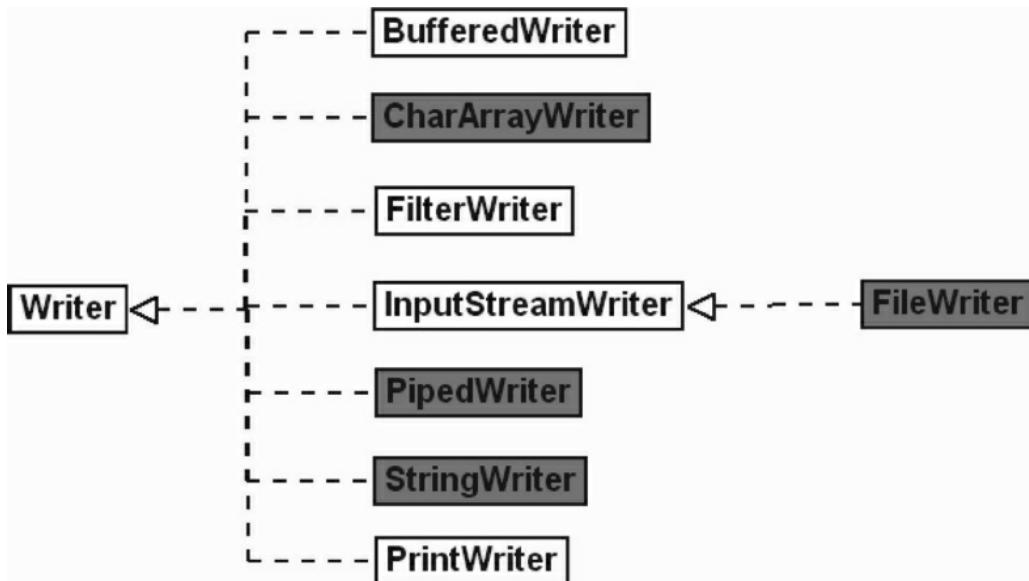


Figura 17.7 - Gerarchia dei Writer.

I programmi dovrebbero utilizzare i character stream (con i reader e i writer) per le informazioni di tipo testuale. Infatti questi permettono di leggere qualsiasi tipo di carattere Unicode.

17.3.2 I Byte Stream

Nel package `java.io` esistono gerarchie di classi parallele ai reader e ai writer che però sono destinate alla lettura e alla scrittura di informazioni non testuali (per esempio file binari come immagini o suoni). Queste classi infatti leggono (e scrivono) dividendo i dati in “pezzi” di 8 bit ognuno. Quindi quanto detto per `Reader` e `Writer` vale anche per le classi `InputStream` e `OutputStream`. Le Figure 17.8 e 17.9 mostrano le gerarchie delle classi principali per quanto riguarda rispettivamente `InputStream` e `OutputStream`, con i decoratori colorati in grigio.

Due di queste classi, `ObjectInputStream` e `ObjectOutputStream`, sono utilizzate per la cosiddetta serializzazione di oggetti, di cui vedremo un esempio nel prossimo paragrafo.

17.3.3 Le superinterfacce principali

`Reader` e `InputStream` definiscono praticamente gli stessi metodi, ma per differenti tipi di dati. Per esempio, `Reader` contiene i seguenti metodi per leggere caratteri ed array di caratteri:

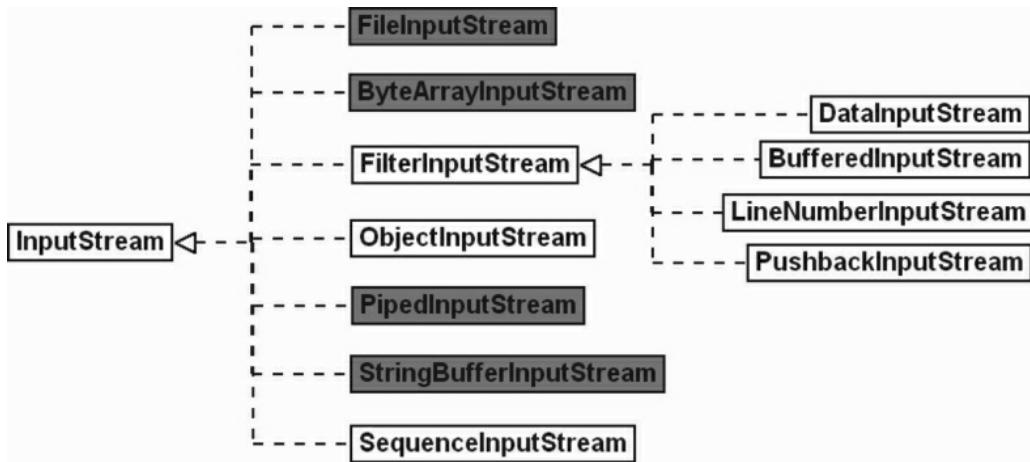


Figura 17.8 - Gerarchia degli InputStream.

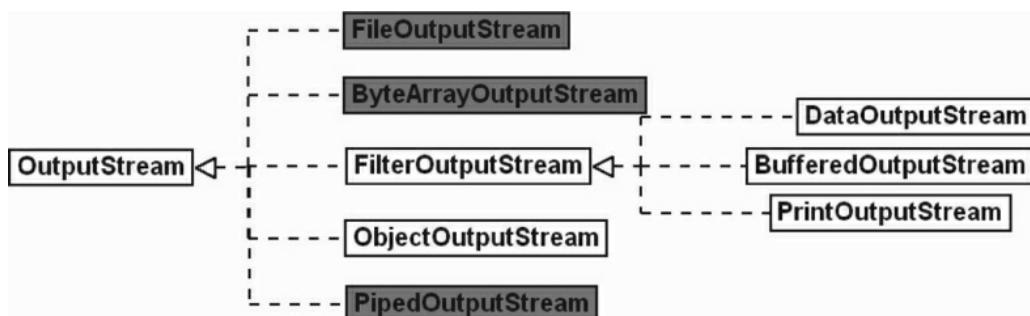


Figura 17.9 - Gerarchia degli OutputStream.

- int read() throws IOException
- int read(char cbuf[]) throws IOException
- int read(char cbuf[], int offset, int length)
- throws IOException

InputStream definisce gli stessi metodi, ma per leggere byte ed array di byte:

- int read() throws IOException
- int read(byte cbuf[]) throws IOException
- int read(byte cbuf[], int offset, int length)
- throws IOException

Inoltre, sia Reader che InputStream forniscono i seguenti metodi d'utilità:

- ❑ `int available() throws IOException`: restituisce il numero di byte che possono essere letti all'interno dello stream.
- ❑ `void close() throws IOException`: rilascia le risorse di sistema associate allo stream (ma non provoca una deallocazione immediata). Nonostante il garbage collector possa implicitamente chiudere uno stream quando non è più possibile referenziare l'oggetto, conviene comunque chiudere gli stream esplicitamente ogni volta che è possibile, per migliorare le prestazioni dell'applicazione.
- ❑ `long skip(long nbytes) throws IOException`: prova a leggere e a scartare `nbytes`. Ritorna il numero di byte scartati.

Anche `Writer` e `OutputStream` sono da considerarsi classi parallele. `Writer` definisce i seguenti metodi per scrivere caratteri ed array di caratteri:

- ❑ `int write(int c)`
- ❑ `int write(char cbuf[])`
- ❑ `int write(char cbuf[], int offset, int length)`

`OutputStream` invece definisce gli stessi metodi ma per i byte:

- ❑ `int write(int c)`
- ❑ `int write(byte cbuf[])`
- ❑ `int write(byte cbuf[], int offset, int length)`

17.3.4 Chiusura degli stream

Tutti gli stream (reader, writer, input stream e output stream) sono automaticamente aperti quando creati. Ogni stream si dovrebbe chiudere esplicitamente chiamando il metodo `close()`. Il garbage collector infatti, non può chiudere uno stream se questo è ancora aperto, e di conseguenza non potrà neanche deallokarne la memoria.

Conviene quindi chiudere gli stream esplicitamente ogni volta che è possibile per migliorare le prestazioni dell'applicazione. Una buona tecnica è chiudere lo stream (una volta che si ha finito di utilizzarlo) nella clausola `finally` di un blocco `try-catch`.

Per esempio consideriamo il seguente esempio che copia un file:

```
import java.io.*;

public class CloseResources {
    public static void copyFile(String source, String destination) throws
        IOException {
        InputStream inputStream = new FileInputStream(source);
        OutputStream outputStream = new FileOutputStream(destination);
        try {
            byte[] byteBuffer = new byte[1024];
            int bytesRead = 0;
            while ((bytesRead = inputStream.read(byteBuffer)) >= 0)
                outputStream.write(byteBuffer, 0, bytesRead);
        } finally {
            inputStream.close();
            outputStream.close();
        }
    }
}
```

```
        outputStream.write(byteBuffer, 0, bytesRead);
    } finally {
        outputStream.close();
        inputStream.close();
    }
}
```

Ma oramai dalla versione 7 di Java possiamo utilizzare il costrutto “try with resources”. Abbiamo già introdotto il concetto nel paragrafo 8.4. Abbiamo detto che molte classi della libreria standard di cui è doverosa una chiusura dopo che sono state usate, sono state riviste per implementare l’interfaccia `Closeable` o `AutoCloseable`. Questo rende tali classi eleggibili come parametri per il “try with resources”. Segue il precedente esempio modificato per supportare tale costrutto:

```
public static void copyFile(String source, String destination) throws
    IOException {
    try (InputStream inputStream = new FileInputStream(source);
        OutputStream outputStream = new FileOutputStream(destination)) {
        byte[] byteBuffer = new byte[1024];
        int bytesRead = 0;
        while ((bytesRead = inputStream.read(byteBuffer)) >= 0)
            outputStream.write(byteBuffer, 0, bytesRead);
    }
}
```

Si può notare una maggiore compattezza del codice da scrivere, e la garanzia di non dimenticarsi di chiudere il flusso.

17.4 Input ed output “classici”

In questa unità didattica verranno presentati alcuni esempi di classiche situazioni di input-output.

17.4.1 Lettura di input da tastiera

Il seguente codice definisce un programma che riesce a leggere ciò che l’utente scrive sulla tastiera e, dopo la pressione del tasto Invio, ristampa quanto letto. Per terminare il programma bisogna digitare “fine” e premere Invio.

Se si esegue quest’applicazione con EJE, per potere iniziare a digitare con la tastiera, bisognerà posizionarsi sull’area di output dell’editor (che infatti si chiama `IOArea`).

```
import java.io.*;

public class KeyboardInput {
    public static void main (String args[]) throws IOException {
```

```
String stringa = null;
System.out.println("Digita qualcosa e premi invio..." + "\nPer terminare
il programma digitare \"fine\"");
try (InputStreamReader ir = new InputStreamReader(System.in);
BufferedReader in = new BufferedReader(ir)) {
    stringa = in.readLine();
    while ( stringa != null ) {
        if (stringa.equals("fine")) {
            System.out.println("Programma terminato");
            break;
        }
        System.out.println("Hai scritto: " + stringa);
        stringa = in.readLine();
    }
}
}
```

Un `InputStreamReader` viene creato ed agganciato all'oggetto `System.in`, che rappresenta l'input di default del nostro sistema (quasi sicuramente la tastiera).

La classe `InputStreamReader` è molto particolare. Consente di trasformare una fonte dati di tipo `byte` in una fonte dati di tipo `char`.

L'oggetto `isr` da solo ci avrebbe già permesso di leggere con uno dei metodi `read()` il contenuto dallo stream proveniente dalla tastiera, ma sarebbe stato molto più complesso realizzare il nostro obiettivo. Infatti avremmo dovuto concretizzare il seguente algoritmo:

1. leggere ogni byte con un ciclo;
2. per ogni carattere letto con il metodo `read()`;
3. controllare se il carattere coincide con “\n” (ovvero il tasto Invio);
4. se è verificato il passo 3, stampare i caratteri precedenti e svuotare lo stream. Se i caratteri precedenti formavano la parola “fine”, terminare l'applicazione. Se la condizione 3 non è verificata, leggere il prossimo carattere.

Si noti come invece, decorando con un `BufferedReader` l'oggetto `isr`, la situazione si sia notevolmente semplificata. Il `BufferedReader` infatti mette a disposizione del programmatore il metodo `readLine()`, che restituisce il contenuto dello stream sotto forma di stringa fino al tasto Invio.

Quanta strada abbiamo dovuto fare con Java per leggere finalmente un carattere da tastiera! L'input-output di Java è effettivamente un argomento complesso, ma come constateremo presto la complessità è costante per qualsiasi operazione di input-output. Per esempio, il metodo `println()` viene utilizzato sia per stampare sul prompt DOS sia per stampare contenuto dinamico in una risposta

HTTP o in un flusso viaggiante in rete (cfr. appendice P on line dedicata al Networking).

Con Java 8 `BufferedReader` si è arricchita del nuovo metodo `lines()`, che restituisce uno stream processato in maniera lazy. Per esempio potremmo leggere da una fonte tutte le righe per trovare la prima riga che contiene la parola “cenzie”:

```
Stream<String> lines = reader.lines();
Optional<String> cenzie =
    lines.filter (s -> s.contains("cenzie")).findFirst();
cenzie.ifPresent(System.out::println);
```

Abbiamo già parlato del metodo `printf()` nel paragrafo 13.1.5 dedicata alle formattazioni di output. Si trova nella classe `PrintStream` (la classe di `System.out`) e basa la sua implementazione sul metodo `format()` della classe `java.util.Formatter`. In Java è ora possibile scrivere il seguente codice:

```
boolean b = true;
System.out.printf("Ecco un valore booleano: %b, ora si formatta "
    + "come in C!", b);
```

ottenendo il seguente output:

```
Ecco un valore booleano: true, ora si formatta come in C!
```

Inoltre la classe `Scanner` del package `java.util` consente di semplificare la lettura di sorgenti di input, siano esse stringhe, tipi primitivi, file o altro. Per esempio, è possibile leggere da tastiera tramite `Scanner` con il seguente codice:

```
try (Scanner sc = new Scanner(System.in)) {
    String testoDigitato = "";
    while (sc.hasNext()) {
        testoDigitato += sc.next();
        System.out.println(testoDigitato);
    }
}
```

`Scanner` può utilizzare anche le espressioni regolari (cfr. paragrafo 13.1.6) per realizzare complesse operazioni di riconoscimento di testo.

Si noti che anche `Scanner`, implementando `Closeable`, è gestibile con il costrutto “try with resources”.

17.4.2 Gestione dei file

Nel package `java.io` esiste la classe `File` che astrae il concetto di file generico.

Anche una directory è un file: un file che contiene altri file.

Seguono i dettagli dei metodi più interessanti di questa classe:

- ❑ `boolean exists()` restituisce `true` se l'oggetto file coincide con un file esistente sul file system.
- ❑ `String getAbsolutePath()` ritorna il path assoluto del file.
- ❑ `String getCanonicalPath()`, come il metodo precedente, restituisce il path assoluto ma senza utilizzare i simboli “.” e “..”.
- ❑ `String getName()` restituisce il nome del file o della directory.
- ❑ `String getParent()` restituisce il nome del file della directory che contiene il file.
- ❑ `boolean isDirectory()` restituisce `true` se l'oggetto file coincide con una directory esistente sul file system.
- ❑ `boolean isFile()` restituisce `true` se l'oggetto file coincide con un file esistente sul file system.
- ❑ `String[] list()` ritorna una array di stringhe contenente i nomi dei file presenti nella directory su cui viene chiamato il metodo. Se questo metodo viene invocato su un file che non è una directory, restituisce `null`.
- ❑ `boolean delete()` tenta di cancellare il file corrente.
- ❑ `long length()` restituisce la lunghezza del file.
- ❑ `boolean mkdir()` tenta la creazione di una directory il cui path è descritto dall'oggetto `File` corrente.
- ❑ `boolean renameTo(File newName)` tenta di rinominare il file corrente. Tale metodo restituisce `true` se ha successo.
- ❑ `boolean canRead()` restituisce `true` se il file o la directory sono leggibili dall'utente corrente (ovvero se ha permesso in lettura).
- ❑ `boolean canWrite()` restituisce `true` se il file o la directory sono modificabili.
- ❑ `boolean createNewFile()` crea un nuovo file vuoto come descritto dall'oggetto corrente se tale file non esiste già. Restituisce `true` se e solo se tale file viene creato.

I costruttori della classe `File` sono i seguenti:

- ❑ `File(String pathname)`
- ❑ `File(String dir, String subpath)`
- ❑ `File(File dir, String subpath)`

Di seguito qualche esempio:

```
File dir = new File("/usr", "local");
File file = new File(dir, "Abc.java");
File dir2 = new File("C:\\\\directory");
File file2 = new File(dir2, "Abc.java");
```

Si noti come nel primo caso abbiamo istanziato una directory e un file su un sistema Unix e nel secondo, invece, abbiamo eseguito le stesse operazioni su un sistema Windows. È possibile utilizzare quindi il separatore per il path dei file per i vari sistemi operativi in maniera dipendente, ma anche utilizzare come separatore “/” pure su sistemi Windows. La migliore idea è però utilizzare la costante statica della classe `File` (dipendente dal sistema operativo): `File.pathSeparator`. Questa varrà su sistemi Windows “\” e su sistemi Unix “/”. Per esempio:

```
File file = new File(.." + File.pathSeparator + "Abc.java");
```

Istanziare un file non significa però crearlo fisicamente sul file system; per farlo è necessario utilizzare gli stream.

Il seguente codice consente di creare una copia di backup di un file specificato da riga di comando:

```
import java.io.*;

public class BackupFile {
    public static void main(String[] args) throws IOException {
        if (args.length == 0) {
            System.out.println("Specificare nome del file!");
            System.exit(0);
        }
        File inputFile = new File(args[0]);
        File outputFile = new File(args[0] + ".backup");
        try (FileInputStream fis = new FileInputStream(inputFile);
             FileOutputStream fos = new FileOutputStream(outputFile)) {
            int b = 0;
            while ((b = fis.read()) != -1) {
                fos.write(b);
            }
            System.out.println("Eseguito backup in " + args[0]
                + ".backup!");
        }
    }
}
```

Il codice precedente ha bisogno che sia specificato come argomento da riga di comando il nome del file di cui eseguire la copia, altrimenti termina stampando un messaggio esplicativo. Il codice è abbastanza semplice. Vengono dapprima istanziati due oggetti `File`: `inputFile`, che rappresenta il file originale, ed `outputFile`, che rappresenta il file copia. Subito dopo vengono istanziati i relativi canali di input e di output verso questi due file, con la creazione dei due oggetti `fis` (di tipo `FileInputStream`) e `fos` (di tipo `FileOutputStream`). Infine, prima di stampare un messaggio di

successo, l'applicazione esegue un ciclo `while` su tutti i byte che possono essere letti tramite l'oggetto `fis`, immagazzinandoli al volo all'interno della variabile `b`, per poi scriverli all'interno di `fos`. Si noti come il ciclo termini solo quando il metodo `read()` del `FileInputStream` restituisce `-1` (ovvero quando non ci sono più byte da leggere).

In questo caso non sono stati utilizzati decoratori. Così abbiamo creato una copia binaria del file copiando i byte nella maniera più generica possibile. Questo fa sì che sia possibile specificare qualsiasi tipologia di file in input per ottenerne una copia perfetta, sia esso un file di testo, un'immagine, un documento Word o qualsiasi altro tipo di file binario.

Nel caso il file da copiare si trovi in una cartella diversa da quella in cui si sta lanciando l'applicazione, è necessario specificare il nome del file comprensivo del suo path relativo o assoluto. Un esempio di istruzione valida per eseguire la precedente applicazione mediante la specifica di un path relativo potrebbe essere il seguente:

```
java BackupFile ..\..\MioFileDaCopiare
```

Un esempio che sfrutta invece un path assoluto, potrebbe essere il successivo:

```
java BackupFile C:\MiaCartella\MioFileDaCopiare
```

La versione 6 di Java ha introdotto anche altri metodi interessanti per la classe `File`:

- ❑ `setWritable(boolean writable)` consente di impostare il permesso in scrittura su un file mediante un semplice booleano.
- ❑ `setWritable(boolean writable, boolean ownerOnly)` consente di impostare il permesso in scrittura su un file mediante un semplice booleano. Inoltre con il secondo argomento booleano è possibile specificare se il permesso in scrittura debba essere accordato solo all'owner del file (valore `true`) o a chiunque (valore `false`). Se il sistema operativo sottostante non è in grado di determinare l'owner del file, il permesso verrà accordato a tutti gli utenti indipendentemente dal valore dell'argomento `ownerOnly`.
- ❑ `setReadable(boolean readable)` consente di impostare il permesso di lettura di un file.
- ❑ `setReadable(boolean readable, boolean ownerOnly)` consente di impostare il permesso di lettura su un file ed inoltre, come per il metodo `setWritable(boolean writable, boolean ownerOnly)`, con il secondo argomento, è possibile specificare se il permesso in lettura debba essere concesso solo all'owner del file o a chiunque. Anche in questo caso, se il sistema operativo sottostante non è in grado di determinare l'owner del file, il permesso verrà accordato a tutti gli utenti indipendentemente dal valore dell'argomento `ownerOnly`.
- ❑ `setExecutable(boolean executable)`, come per i metodi `setWritable(boolean writable)` e `setReadable(boolean readable)` consente di impostare il permesso di

esecuzione di un file.

- `setExecutable(boolean executable, boolean ownerOnly)` funziona come con i metodi `setWritable(boolean writable, boolean ownerOnly)` e `setReadable(boolean readable, boolean ownerOnly)`.
- `getTotalSpace()` restituisce la dimensione in byte complessiva della partizione astratta dal nome del file. Restituisce il valore `0L` se il nome del file non coincide con una partizione valida del sistema operativo.
- `getFreeSpace()` restituisce la dimensione in byte dello spazio libero della partizione astratta dal nome del file. In particolare restituisce il numero di byte non allocati al momento della chiamata. Questo metodo non può però prevedere se, dopo una frazione di secondo dalla sua invocazione, un altro processo (magari esterno alla Virtual Machine) andrà ad allocare nuovi byte della partizione in questione. La sua precisione è quindi spesso relativa. Anche questo metodo restituisce il valore `0L` se il nome del file non coincide con una partizione valida del sistema operativo.
- `getUsableSpace()` restituisce la dimensione in byte dello spazio realmente utilizzabile della partizione astratta dal nome del file. Questo metodo è più affidabile rispetto a `getFreeSpace()` per quanto riguarda una stima dello spazio realmente utilizzabile. Infatti controlla anche eventuali permessi in scrittura dei file ed altre restrizioni che il sistema operativo potrebbe definire. Neanche questo metodo può però prevedere se, dopo una frazione di secondo dalla sua invocazione, un altro processo andrà ad allocare nuovi byte nella stessa partizione. Anche la sua precisione è dunque spesso relativa. Ed anche questo metodo restituisce il valore `0L` se il nome del file non coincide con una partizione valida del sistema operativo.

17.4.3 Serializzazione di oggetti

Con “serializzazione di oggetti” intendiamo il processo per rendere persistente un oggetto Java. Rendere “persistente” un oggetto significa far sopravvivere l’oggetto oltre lo shutdown del programma. Solitamente questo significa salvare lo “stato dell’oggetto”, ovvero le variabili d’istanza con i relativi valori, all’interno di un file (o all’interno di un database, ma questo sarà argomento del prossimo modulo).

In Java è possibile serializzare oggetti a patto che implementino l’interfaccia `Serializable` del package `java.io`. Tale interfaccia non contiene metodi ma serve solo a distinguere ciò che è serializzabile da ciò che non lo è. Esistono per esempio classi della libreria standard che, per come sono concepite, non possono essere serializzate. Un esempio potrebbe essere la classe `Thread`. Un thread non ha uno stato, rappresenta un concetto dinamico, un “processo che esegue codice su dati” e non può essere serializzato. La classe `Thread`, infatti, non implementa l’interfaccia `Serializable`. Altri esempi di classi non serializzabili sono tutte le classi di tipo `Stream`.

Ora, tenendo presente che serializzare un oggetto significa salvare il suo stato interno, ovvero salvare il valore delle proprie variabili d’istanza, se vogliamo creare una classe da cui istanziare oggetti da serializzare, bisogna stare attenti alle variabili d’istanza. Se infatti una delle variabili d’istanza è di tipo `Thread` o di tipo `Writer`, provando a serializzare l’oggetto otterremmo al runtime

una `java.io.NotSerializableException`. La stessa eccezione scatterebbe nel caso in cui la nostra classe non implementasse `Serializable`.

Per ovviare a questo problema esiste il modificatore `transient`. Esso può essere anteposto solo a variabili d'istanza, ed avverterà al runtime la JVM che la variabile marcata `transient` non deve essere serializzata. È obbligatorio quindi marcare `transient` le variabili non serializzabili (per esempio di tipo `Thread`), tuttavia potremmo anche desiderare non serializzare volutamente alcune variabili per ragioni di sicurezza.

Come esercizio consideriamo la seguente semplice classe da serializzare:

```
public class Persona implements java.io.Serializable {
    private String nome;
    private String cognome;
    private transient Thread t = new Thread();
    private transient String codiceSegreto;

    public Persona(String nome, String cognome, String cs) {
        this.setNome(nome);
        this.setCognome(cognome);
        this.setCodiceSegreto(cs);
    }
    //Metodi set e get omessi
    public String toString() {
        return "Nome: " + getNome() + "\nCognome: " + getCognome() +
               "\nCodice Segreto: " + getCodiceSegreto();
    }
}
```

Abbiamo marcato `transient` una variabile `Thread` (eravamo obbligati) e la variabile `codiceSegreto` di tipo `String`. Ovviamente serializzare su un file il codice segreto potrebbe essere una mossa poco previdente...

Per serializzare l'oggetto basta eseguire la seguente classe:

```
import java.io.*;

public class SerializeObject {
    public static void main (String args[]) throws IOException {
        Persona p = new Persona ("Bruno", "Montagna", "xxx");
        try (FileOutputStream f =
              new FileOutputStream (new File("persona.ser"));
              ObjectOutputStream s = new ObjectOutputStream (f);) {
            s.writeObject (p);
            System.out.println("Oggetto serializzato!");
        }
    }
}
```

Il codice è semplice, non molto diverso dagli altri esempi che finora abbiamo visto. Le particolarità

stanno nella decorazione eseguita con l'oggetto di tipo `ObjectOutputStream` e nell'utilizzo del metodo `writeObject()`.

Per deserializzare l'oggetto basta eseguire la seguente classe:

```
import java.io.*;

public class DeSerializeObject {
    public static void main (String args[]) throws Exception {
        Persona p = null;
        try (FileInputStream f =
              new FileInputStream (new File("persona.ser"));
             ObjectInputStream s = new ObjectInputStream (f);) {
            p = (Persona)s.readObject();
            System.out.println("Oggetto deserializzato!");
            System.out.println(p);
        }
    }
}
```

di cui di seguito eccone l'output:

```
Oggetto deserializzato!
Nome: Bruno
Cognome: Montagna
Codice Segreto: null
```

Anche le variabili dichiarate `static` non saranno serializzate. Infatti non sarebbe giusto cambiare il valore di una variabile statica, che è condivisa da tutte le istanze della classe, solo perché è possibile deserializzare una particolare istanza. Le variabili statiche sono implicitamente dichiarate `transient`.

Attenzione! Esiste uno strano vincolo per realizzare con successo una deserializzazione. Bisogna inserire un costruttore senza argomenti nella prima superclasse della classe da serializzare che non implementa `Serializable`, altrimenti andremo incontro ad una `java.io.InvalidClassException`, con un messaggio:

```
no valid constructor
```

quando verrà invocato il metodo `readObject()`.

Interessante è concludere il discorso iniziato nel paragrafo 11.1.2 sulla clonazione di oggetti. Avevamo visto che il metodo `clone()` solitamente si limita a creare una *shallow copy* dell'oggetto, e che avremmo dovuto riscrivere il metodo `clone()` in maniera molto costosa se avessimo voluto ottenere una *deep copy*. Per gli oggetti `Serializable` però, c'è una scorciatoia interessante. Il seguente metodo:

```

public static <O extends Serializable> O cloneObject(O object)
throws IOException, ClassNotFoundException {
    ByteArrayOutputStream bout = new ByteArrayOutputStream();
    ObjectOutputStream out = new ObjectOutputStream(bout);
    out.writeObject(object);
    out.close();
    ByteArrayInputStream bin =
        new ByteArrayInputStream(bout.toByteArray());
    ObjectInputStream in = new ObjectInputStream(bin);
    O objectCloned = (O) in.readObject();
    in.close();
    return objectCloned;
}

```

Prende in input un oggetto serializzabile e ne ritorna una copia profonda, in quanto essa viene creata direttamente tramite stream, e non impostando ad uno ad uno i campi. Questo tipo di implementazione inoltre non ha bisogno di essere aggiornata quando si aggiornano gli oggetti da clonare.

Finalmente dopo 19 anni sono state introdotte due classi standard per l'encoding e il decoding (in italiano codifica e decodifica) in Base64. Siamo parlando della codifica che trasforma un array di byte in una stringa. Questo tipo di encoding è molto usato per inserire dati binari in una email, o per passarli tramite protocolli di rete basati su stringhe come l'HTTP. Sino ad ora siamo stati obbligati ad utilizzare implementazioni non ufficiali come sun.misc.BASE64Encoder, ma ora possiamo finalmente usare le classi Base64.Encoder e Base64.Decoder del package java.util.

17.5 NIO 2.0

Con Java 7 sono state introdotte molte novità sulla gestione dei file. Già dalla versione 1.4 di Java fu introdotto un nuovo package: `java.nio` (“nio” sta per “New Input Output”). Questo definisce nuove classi e interfacce per gestire più efficacemente l’input output in Java. In particolare il nuovo modello proponeva l’uso di buffer (contenitori di dati da passare in input o ricevere in output) e channel (connessioni ad entità capaci di eseguire operazioni di input output). Per esempio il seguente codice legge un file di testo, lo inserisce in un `ByteBuffer` per poi rileggerlo e stamparlo a video tramite un oggetto di tipo `FileChannel`:

```

String path = "C:\\myfile.txt";
try (FileInputStream fis = new FileInputStream(path)) {
    FileChannel fileChannel = fis.getChannel(); {
        ByteBuffer buffer = ByteBuffer.allocate(128);
        int bytesRead = fileChannel.read(buffer);
        while (bytesRead != -1) {
            System.out.println("Letto: " + bytesRead);
            //converte il buffer da buffer di scrittura a buffer di lettura
        }
    }
}

```

```

        buffer.Aip();
        // Stampa byte letti
        while (buffer.hasRemaining()) {
            System.out.print((char) buffer.get());
        }
        buffer.clear();
        bytesRead = fileChannel.read(buffer);
    }
} catch (IOException e) {
    e.printStackTrace();
}

```

Il codice è abbastanza chiaro, visto che non si distacca molto da quello che abbiamo visto sino ad ora. I channel rispetto agli oggetti stream hanno la possibilità di gestire più situazioni, come per esempio quando si ha a che fare con file protetti o si vuole utilizzare stream non bloccanti (ovvero che non si bloccano se aspettano un input). I channel non sono stati introdotti con lo scopo di sostituire gli stream. La libreria NIO semplicemente consente operazioni più complesse che la libreria IO non permette. Il problema con la prima versione di NIO era la curva di apprendimento. Ora con NIO 2.0 abbiamo anche una notevole semplicità di utilizzo. È probabile che questa semplicità farà in modo che nel corso del tempo NIO 2.0 diventerà il modello preferito degli sviluppatori a discapito del vecchio modello.

In particolare sono stati introdotti quattro nuovi package in Java 7: `java.nio.file`, `java.nio.file.spi`, `java.nio.file.attribute`, `java.nio.file.spi`. Le novità più importanti risiedono essenzialmente nell'introduzione delle classe `Files` e l'interfaccia `Path` (e le sue implementazioni), che di fatto ora forniscono metodi più semplici ed efficienti per effettuare operazioni su file. Effettivamente l'utilizzo di queste nuove strutture dovrebbe sostituire l'utilizzo della classe `java.io.File`. A scopo di facilitarne l'interoperabilità dei modelli (e volendo la migrazione) è stato creato il metodo `toPath()` nella classe `File` per ottenere un oggetto `Path`. Esiste anche l'omologo metodo `toFile()` nell'interfaccia `Path`.

17.5.1 L'interfaccia Path

`Path` rappresenta un percorso sul file system che astrae il concetto di file o di directory. Un path (in inglese “percorso”) è un concetto familiare per chi usa il computer. Un tipico esempio di `Path` potrebbe essere: “C:\Windows\write.exe” che rappresenta il programma WordPad su Windows. Su Linux per esempio un `Path` è qualcosa di diverso visto il differente file system: “/etc/grub.conf” identifica il file di configurazione del programma Grub. Questa è anche la ragione per cui `Path` è un’interfaccia e non una classe. Per ottenere un’implementazione di `Path` infatti bisogna utilizzare l’oggetto `FileSystem`:

```
Path path1 = FileSystems.getDefault().getPath("/root/aFile.txt");
```

o più semplicemente l’oggetto `Paths`:

```
Path path2 = Paths.get("C:\\Program Files\\EJE");
```

Il doppio backslash “\\” è obbligatorio visto che uno solo serve per identificare i caratteri di escape.

Una volta ottenuto un oggetto Path è possibile utilizzare una serie di metodi potenti e semplici da utilizzare. Segue qualche esempio:

```
Path pathToDesktop = Paths.get("C:\\Users\\user\\Desktop");
Path pathToDocuments = Paths.get("C:\\Users\\user\\Documents");
System.out.println("toString: " + pathToDesktop.toString());
System.out.println("getFileName: " + pathToDesktop.getFileName());
System.out.println("getName(0): " + pathToDesktop.getName(0));
System.out.println("getNameCount: " + pathToDesktop.getNameCount());
System.out.println("subpath(0,2): " + pathToDesktop.subpath(0,2));
System.out.println("getRoot: " + pathToDesktop.getRoot());
System.out.println("getParent: " + pathToDesktop.getParent());
System.out.println("toUri: " + pathToDesktop.toUri());
System.out.println("path from p1 to p2: "
    + pathToDesktop.relativize(pathToDocuments));
System.out.println("path from p2 to p1: "
    + pathToDocuments.relativize(pathToDesktop));
System.out.println("pathToDesktop.equals(pathToDocuments): "
    + pathToDesktop.equals(otherPath));
System.out.println("pathToDesktop.startsWith: "
    + pathToDesktop.startsWith(pathToDocuments.subPath(0,2)));
System.out.println("pathToDesktop.endsWith: "
    + pathToDesktop.endsWith(pathToDocuments.subPath(0,2))));
```

Segue l'output del codice presentato:

```
toString: C:\\Users\\user\\Desktop
getFileName: Desktop
getName(0): Users
getNameCount: 3
subpath(0,2): Users\\user
getRoot: C:\\
getParent: C:\\Users\\user
toUri: file:///C:/Users/user/Desktop
path from p1 to p2: ..\\Documents
path from p2 to p1: ..\\Desktop
pathToDesktop.equals(pathToDocuments): false
pathToDesktop.startsWith: false
pathToDesktop.endsWith: false
```

Alcuni di questi metodi non hanno bisogno di ulteriori commenti, quindi commenteremo solo i più “misteriosi”. Il metodo `subpath()` per esempio restituisce il numero di elementi che compongono il path, escludendo il nodo radice (nell'esempio è “C:\\” ma per un sistema Unix-Linux sarebbe “/”). Il

metodo `getRoot()` serve proprio per identificare il nodo radice. Il metodo `getParent()` invece restituisce la cartella in cui è contenuto il file rappresentato dall'oggetto `Path`. Il metodo `toUri()` restituisce il path in formato URI (Uniform Resource Identifier). Interessante il metodo `relativize()` che restituisce il percorso che serve per andare dal path su cui si chiama il metodo al path che viene passato come parametro.

Nell'esempio il metodo `startsWith()` ritorna `false` in quanto il metodo `subpath()` non restituisce il nodo radice.

17.5.2 La classe Files

Per compiere operazioni sui file la classe principale di riferimento ora si chiama `Files` (come `Path` è localizzata nel package `java.nio.file`). Esistono tantissimi metodi di utilità nella classe `Files` (sono oltre 50 escludendo quelli ereditati da `Object`, tutti statici). Anche in questo caso i metodi sono piuttosto intuitivi. Per esempio per ottenere un `BufferedWriter` per scrivere su un file avremo a disposizione il metodo `newBufferedWriter()`. Ecco un esempio che fa uso del costrutto `try with resources`:

```
Charset charset = Charset.forName("UTF-8");
String contenutoDelFile = "Ciao";
Path path = Paths.get("C:\\\\Users\\\\user\\\\Desktop\\\\test.txt");
try (BufferedWriter writer = Files.newBufferedWriter(path, charset)) {
    writer.write(contenutoDelFile, 0, contenutoDelFile.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Come per l'interfaccia `Path` diamo uno sguardo ai metodi più interessanti della classe `Files`:

```
Path directory = Paths.get("C:\\\\Users\\\\user\\\\Desktop");
Path file = Paths.get("C:\\\\Users\\\\user\\\\Desktop\\\\test.txt");
System.out.println("Files.exists(directory): " +
    + Files.exists(directory));
System.out.println("Files.isReadable(file): " + Files.isReadable(file));
System.out.println("Files.isWritable(file): " + Files.isWritable(file));
System.out.println("Files.isExecutable(file): " +
    + Files.isExecutable(file));
System.out.println("Files.isSameFile(file): " +
    Files.isSameFile(directory, file));
```

e di seguito l'output:

```
Files.exists(directory): true
Files.isReadable(file): true
Files.isWritable(file): true
Files.isExecutable(file): true
Files.isSameFile(file): false
```

Come si può immaginare il metodo `exists()` controlla se il file specificato esiste fisicamente. Solleva una `SecurityException` nel caso il file non sia accessibile. Esiste anche il metodo `notExists()`.

I metodi `isReadable()`, `isWritable()` e `isExecutable()` servono rispettivamente per capire i permessi di leggibilità, scrittura ed esecuzione del file.

Il metodo `isSameFile()` invece restituisce `true` se e solo se i file specificati come parametri puntano allo stesso file (cosa possibile per esempio in presenza di file di tipo collegamento).

I metodi che consentono operazioni sui file sono estremamente intuitivi nonostante siano anche molto potenti. Per la cancellazione di un file esistono due metodi: `delete()` e `deleteIfExists()`. Entrambi definiscono come parametro in input un oggetto `Path` che rappresenta il file da cancellare. La differenza tra questi due metodi è che il secondo non solleva eccezioni nel caso la cancellazione fallisca. Il metodo `delete()` invece potrebbe fallire nel caso ci sia un problema con i permessi di cancellazione del file (`IOException`), se si provasse a eliminare un file che non esiste (`NoSuchFileException`) oppure se si provasse a eliminare una directory non vuota (`DirectoryNotEmptyException`).

Per quanto riguarda la copia di file il metodo `copy()` prende in input due parametri `Path` (il primo è il `Path` che si deve copiare mentre il secondo è quello di destinazione) e un varargs di tipo `CopyOption`. Quest'ultima è un'interfaccia implementata da due enumerazioni `StandardCopyOption` e `LinkOption`. Ecco un semplice esempio:

```
import static java.nio.file.StandardCopyOption.*;
import java.nio.file.LinkOption;
...
Path source = Paths.get("C:\\\\Users\\\\user\\\\Desktop\\\\test.txt");
Path target = Paths.get("C:\\\\Users\\\\user\\\\Desktop\\\\copy.txt");
Files.copy(source, target, REPLACE_EXISTING, COPY_ATTRIBUTES,
           LinkOption.NOFOLLOW_LINKS);
```

che crea una copia del file `test.txt` chiamandola `copy.txt`. Delle tre opzioni specificate come varargs, i primi due sono elementi importati staticamente dall'enumerazione `StandardCopyOption`, mentre la terza è un elemento dell'enumerazione `LinkOption`. `REPLACE_EXISTING` implica che se il file `copy.txt` esistesse già, sarà sovrascritto. `COPY_ATTRIBUTES` infine copia anche gli attributi del file `test.txt`; per esempio se il file `test.txt` era in sola lettura anche il file `test.txt` sarà in sola lettura. `NOFOLLOW_LINKS` farà in modo che, nel caso il file `test.txt` fosse un collegamento ad un file, non sia copiato il file vero e proprio ma solo il link. Infatti di default questo metodo avrebbe copiato il file di destinazione e non il link. Queste `CopyOption` sono le uniche supportate dal metodo `copy()`.

È possibile copiare directory ma i file contenuti in essa non saranno copiati. È bene notare che però esistono anche due overload del metodo `copy()` che interagiscono con `InputStream` e `OutputStream`. Volendo quindi è possibile usare un `OutputStream` per scrivere una directory dopo averla letta. Inoltre esiste un meccanismo basato sull'interfaccia `FileVisitor` che consente di fare operazioni ricorsive (e quindi anche

quella di fare copie ricorsivamente).

Il discorso è molto simile per lo spostamento dei file. Anche il metodo `move()` ha la stessa firma di parametri di `copy()`. La differenza sta nel fatto che `move()` supporta solo gli elementi dell'enumerazione `StandardCopyOption.REPLACE_EXISTING` che abbiamo già descritto, e `StandardCopyOption.ATOMIC_MOVE`. Quest'ultimo serve a garantire che lo spostamento sia eseguito come operazione atomica a tutte quelle che si possono collegare (per esempio è possibile associare ad una directory azioni che vengono eseguite in base a notifiche che sono fatte scattare da eventi; vedi documentazione di `FileVisitor` e `SimpleFileVisitor`).

La classe `Files` contiene anche una quindicina di metodi per gestire gli attributi e i metadati dei file. Per esempio `isHidden()` restituisce `true` se il `Path` specificato in input è un file nascosto, mentre `size()` restituisce la dimensione del `Path` specificato. Ci sono metodi per leggere gli attributi (`getAttribute()` e `readAttributes()`), per scrivere gli attributi (`setAttribute()`), per conoscere o impostare la data dell'ultima modifica (`getLastModifiedTime()` e `setLastModifiedTime()`), conoscere o impostare l'autore (`getOwner()` e `setOwner()`) etc.

Infine la classe `Files` definisce diversi metodi per scrivere e leggere file. Molti di questi prendono in input come parametro opzionale uno o più elementi dell'enumerazione `StandardOpenOption`. Ecco l'elenco e la descrizione dei vari elementi:

- ❑ **WRITE:** apre il file con accesso in scrittura.
- ❑ **APPEND:** aggiunge nuovi dati alla fine del file. Questa opzione viene usata in congiunzione alle opzioni `WRITE` o `CREATE`.
- ❑ **TRUNCATE_EXISTING:** tronca il file a zero byte. Questa opzione viene usata in congiunzione all'opzione `WRITE`.
- ❑ **CREATE_NEW:** crea un nuovo file e solleva un'eccezione se il file esiste già.
- ❑ **CREATE:** apre il file se esiste già o lo crea se non esiste.
- ❑ **DELETE_ON_CLOSE:** cancella il file quando lo stream viene chiuso. Utile nel caso di file temporanei.
- ❑ **SPARSE:** suggerisce (visto che è un'operazione che dipende dal file system) che il file appena creato sia in formato "sparso". Questa opzione avanzata è supportata su alcuni filesystem come NTFS, dove file di grandi dimensioni vengono memorizzati in aree di memoria non contigue che sono comunque disponibili per l'utilizzo.
- ❑ **SYNC:** mantiene il file (sia di contenuti e metadati) sincronizzato con il dispositivo di storage sottostante.
- ❑ **DSYNC:** mantiene il contenuto del file sincronizzato con il dispositivo di storage sottostante.

Metodi adatti a leggere file di piccole dimensioni sono `readAllBytes()`, per leggere file binari, e `readAllLines()`, per leggere file di testo. L'equivalente metodo per la scrittura è `write()`, di cui esistono vari overload per prendere in input o un array di byte, o un oggetto che implementi una collezione o un array di caratteri (o stringhe). Sia quando si legge che quando si scrive un file di

testo, bisogna specificare anche un oggetto Charset per la decodifica (per esempio Charset.forName ("UTF-8")).

Mediante un oggetto Files possiamo ottenere anche BufferedReader e BufferedWriter utilizzando i metodi newBufferedReader() e newBufferedWriter(). Segue un esempio:

```
String s = "Java 7";
try (BufferedWriter writer = Files.newBufferedWriter(file,
            Charset.forName("UTF-8"))) {
    writer.write(s, 0, s.length());
}
```

Inoltre è possibile anche creare stream più semplici per poi decorarli come abbiamo visto precedentemente:

```
import static java.nio.file.StandardOpenOption.*;

Path file = Paths.get("C:\\Program Files\\EJE\\LEGGIMI.htm");
String firma = "Creato da Claudio...";
byte data[] = firma.getBytes();

try (OutputStream out =
      new BufferedOutputStream(
          Files.newOutputStream(file,CREATE, APPEND))) {
    out.write(data, 0, data.length);
}
```

Il codice precedente aggiunge una “firma” al file specificato.

È possibile anche creare oggetti ByteChannel con il metodo newByteChannel() per poi poterli utilizzare sfruttando oggetti ByteBuffer. È semplice anche creare file temporanei con un codice simile al seguente:

```
try {
    Path fileTemporaneo = Files.createTempFile(null, ".tmp");
    System.out.format("Creato il file temporaneo: " + fileTemporaneo);
}
```

Con il codice precedente abbiamo creato un file temporaneo senza specificarne il nome, ma specificandone il suffisso (“.tmp”), e poi abbiamo stampato il suo nome.

Il percorso di salvataggio dipende dal sistema operativo.

Segue l’output:

```
Creato il file temporaneo:
C:\\Users\\user\\AppData\\Local\\Temp\\2041268323571170425.tmp
```

NIO 2.0 offre supporto anche a file ad accesso casuale (`RandomAccessFile`), alla gestione delle directory comprendente la ricorsione sui file e la ricerca tramite sintassi speciali come le regular expression ed espressioni GLOB (una specie di regular expression semplificata per file systems). Inoltre esistono classi e metodi di utilità per gestire al meglio i collegamenti ai file e alla supervisione ad eventi di file e cartelle (`FileVisitor`). Infine se foste interessati a gestioni avanzate di concorrenza basata sull'input output potrete dare uno sguardo alla documentazione dell'interfaccia `Selector`.

In Java 8 il nuovo metodo `lines()` della classe `Files`, che abbiamo già visto con la classe `BufferedReader`, ritorna uno `Stream<String>`. Inoltre `Files` ora ha un metodo che si chiama `list()` e che ritorna uno `Stream<Path>` di file presenti nella directory specificata. Per esempio con il seguente codice:

```
Path pathToDirectory = Paths.get(".");
try (Stream<Path> files = Files.list(pathToDirectory)) {
    files.filter(p -> p.getFileName().toString().endsWith(".java")) .
        forEach(System.out::println);
}
```

stamperemo tutti i file con suffisso `.java` della cartella corrente.

Il metodo `list()` però si limita ai file della directory indicata. Volendo scendere all'interno delle cartelle contenute nella cartella specificata, è possibile usare il metodo `walk()` che prende in input il path e la profondità (depth in inglese) che indica il livello di profondità in cui si deve scendere nell'albero delle directory. Se poi vogliamo anche accedere agli attributi dei file è necessario utilizzare il metodo `find()`. Per esempio con il seguente frammento di codice stampiamo tutti i file creati nella cartella corrente nell'ultima mezz'ora.

```
Path directory = Paths.get(".");
Instant ieri = Instant.now().minus(30, ChronoUnit.MINUTES);
try (Stream<Path> files = Files.find(directory, 1,
    (path, attributi) -> attributi.creationTime().toInstant()
    .compareTo(ieri) >= 0)) {
    files.forEach(System.out::println);
}
```

Riepilogo

In questo modulo abbiamo essenzialmente parlato della comunicazione delle nostre applicazioni con l'esterno. Abbiamo visto come un package complesso come `java.io` sia governato dai rapporti tra classi definiti dal **pattern Decorator**. Abbiamo quindi cercato di dare un'idea dell'utilità di tale pattern e lo abbiamo riconosciuto all'interno del package. Inoltre sono stati forniti esempi per le problematiche di input-output più comuni come l'**accesso ai file**, la **lettura da tastiera** e la **serializzazione di oggetti**.

Abbiamo poi affrontato la descrizione del nuovo modello di Java 7 per l'input output denominato **NIO 2.0**. Abbiamo visto come la maggior parte delle nuove funzionalità sono concentrate

essenzialmente nell'utilizzo della classe `Files` e dell'interfaccia `Path` e ne abbiamo esaltato la semplicità.

Esercizi modulo 17

Esercizio 17.a) Input - Output, Vero o Falso:

1. Il pattern Decorator permette di implementare una sorta di ereditarietà dinamica. Questo significa che, invece di creare tante classi quante sono le entità da astrarre, al runtime sarà possibile concretizzare uno di questi concetti direttamente con un oggetto.
2. I reader e i writer permettono di leggere e scrivere caratteri. Per tale ragione sono detti Character Stream.
3. All'interno del package `java.io` l'interfaccia `Reader` ha il ruolo di `ConcreteComponent`.
4. All'interno del package `java.io` l'interfaccia `InputStream` ha il ruolo di `ConcreteDecorator`.
5. Un `BufferedWriter` è un `ConcreteDecorator`.
6. Gli stream che possono realizzare una comunicazione direttamente con una fonte o una destinazione vengono detti “node stream”.
7. I node stream di tipo `OutputStream` possono utilizzare il metodo:

```
int write(byte cbuf[])
```

per scrivere su una destinazione.

8. Il seguente oggetto `in`:

```
BufferedReader in =  
    new BufferedReader(new InputStreamReader(System.in));
```

permette di usufruire di un metodo `readLine()` che leggerà frasi scritte con la tastiera delimitate dalla battitura del tasto Invio.

9. Il seguente codice:

```
File outputFile = new File("pippo.txt");
```

crea un file di testo chiamato `pippo.txt` nella cartella corrente.

10. Non è possibile decorare un `FileReader`.

Esercizio 17.b) Serializzazione, Vero o Falso:

1. Lo stato di un oggetto è definito dal valore delle sue variabili d'istanza (in un certo istante).
2. L'interfaccia `Serializable` non ha metodi.

3. `transient` è un modificatore applicabile a variabili e classi. Una variabile `transient` non viene serializzata con le altre variabili; una classe `transient` non è serializzabile.
4. `transient` è un modificatore applicabile a metodi e variabili. Una variabile `transient` non viene serializzata con le altre variabili; un metodo `transient` non è serializzabile.
5. Se si provasse a serializzare un oggetto che possiede tra le sue variabili d'istanza una variabile di tipo `Reader` dichiarata `transient`, otterremmo un `NotSerializableException` al runtime.
6. Una variabile `static` non verrà serializzata.
7. Una variabile d'istanza di tipo `OutputStream` deve essere dichiarata `transient` affinché sia coinvolta in una serializzazione.
8. Non è possibile usare come parametro di un “try with resources” un’implementazione di `Serializable`.
9. È possibile creare un clone di un oggetto `transient` con metodi di input output.
10. Un oggetto può essere serializzato anche in Base64.

Esercizio 17.c) New Input Output, Vero o Falso:

1. NIO 2.0 sostituisce del tutto il modello di input output definito con il package `java.io`.
2. L’utilizzo della classe `File` potrebbe essere completamente rimpiazzato dall’utilizzo della classe `Files` e dell’interfaccia `Path`.
3. Il metodo `toPath()` della classe `File` restituisce l’oggetto `Path` equivalente.
4. `Path` è un’interfaccia perché la sua implementazione dipende dalla piattaforma.
5. Il metodo `relativize()` appartiene alla classe `Files`, e restituisce il percorso per arrivare dal `Path` specificato come primo argomento al `Path` specificato come secondo argomento.
6. Il metodo `subPath()` dell’interfaccia `Path` non restituisce il nodo radice.
7. Il metodo `delete()` dell’interfaccia `Path` solleverà un’eccezione nel caso si tenti di cancellare una directory non vuota.
8. Non ha senso ottenere `Reader` o `Writer` da un oggetto `Files`.
9. Il nome del file temporaneo è sempre stabilito dal sistema operativo.
10. Un file temporaneo viene salvato in una directory che dipende dal sistema operativo.

Soluzioni esercizi modulo 17

Esercizio 17.a) Input - Output, Vero o Falso:

- 1. Vero.**
- 2. Vero.**
- 3. Falso.**
- 4. Falso.**
- 5. Vero.**
- 6. Vero.**
- 7. Vero.**
- 8. Vero.**
- 9. Falso.**
- 10. Falso.**

Esercizio 17.b) Serializzazione, Vero o Falso:

- 1. Vero.**
- 2. Vero.**
- 3. Falso.**
- 4. Falso.**
- 5. Falso.**
- 6. Vero.**
- 7. Vero.**
- 8. Falso.**
- 9. Vero.**
- 10. Vero.**

Esercizio 17.c) New Input Output, Vero o Falso:

- 1. Vero.**
- 2. Vero.**
- 3. Vero.**
- 4. Vero.**
- 5. Falso.**
- 6. Vero.**

- 7. Vero.**
- 8. Falso.**
- 9. Falso.**
- 10. Vero.**

Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Aver compreso il pattern Decorator (unità 17.1, 17.2)	<input type="checkbox"/>	
Saper riconoscere nelle classi del package <code>java.io</code> i ruoli definiti nel pattern Decorator (unità 17.3)	<input type="checkbox"/>	
Capire le fondamentali gerarchie del package <code>java.io</code> (unità 17.3)	<input type="checkbox"/>	
Avere confidenza con i tipici problemi che si incontrano con l'input-output, come la serializzazione degli oggetti e la gestione dei file (unità 17.4)	<input type="checkbox"/>	
Avere confidenza con il nuovo modello per l'input output di Java 7 denominato NIO 2.0 (unità 17.5)	<input type="checkbox"/>	

Note:

Java Database Connectivity

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Saper spiegare la struttura dell’interfaccia JDBC (unità 18.1, 18.2).
- ✓ Saper scrivere codice che si connetta a qualsiasi tipo di database (unità 18.2, 18.3).
- ✓ Saper scrivere codice che aggiorna, interroghi e gestisca i risultati qualsiasi sia il database in uso (unità 18.2, 18.3).
- ✓ Saper spiegare le tipiche caratteristiche avanzate di JDBC, come stored procedure, statement parametrizzati e transazioni (unità 18.3).
- ✓ Comprendere le differenze tra le varie versioni delle specifiche JDBC (unità 18.4).

Una delle difficoltà con cui si scontrano tutti i giorni gli sviluppatori Java è doversi confrontare anche con altre aree dell’informatica. Infatti l’apertura di Java verso le altre tecnologie è totale, e quindi bisogna spesso imparare altri linguaggi. Oggi un programmatore italiano medio conosce almeno le basi dei protocolli di rete, un minimo di XML (a cui abbiamo dedicato l’appendice N) e sa interrogare ed aggiornare database tramite SQL. L’SQL (che è l’acronimo di “Structured Query Language”) è il linguaggio standard per l’interrogazione (e non solo) dei database relazionali. È un linguaggio “leader” nato nel 1974 e dopo tanti anni non ha ancora trovato un concorrente. Per quanto oggi si stiano affermando anche altre tecnologie alternative per immagazzinare dati, la quasi totalità delle applicazioni moderne si interfacciano con un database. Questo modulo non spiegherà il linguaggio SQL (on line è possibile trovare una quantità sterminata di tutorial), ma di come Java possa interagire con un database utilizzando l’SQL.

18.1 Introduzione a JDBC

JDBC viene spesso inteso come l’acronimo di “Java Database Connectivity”. Si tratta dello strato di astrazione software che permette alle applicazioni Java di connettersi a database. La potenza, la semplicità, la stabilità e le prestazioni delle interfacce JDBC, in questi anni hanno oramai fatto nascere uno standard. JDBC, rispetto ad altre soluzioni, consente ad un’applicazione di accedere a diversi database senza dover essere modificata. Ciò significa che ad un’applicazione Java, di per sé indipendente dalla piattaforma, possa essere aggiunta anche l’indipendenza dal DBMS (Data Base Management System).

Caliamoci in uno scenario: supponiamo che una società crei un’applicazione Java che utilizza un RDBMS (Relational DBMS) come DB2, lo storico prodotto della IBM. La stessa applicazione esegue su diverse piattaforme come server Solaris e client Windows e Linux. Ad un certo punto, per strategie aziendali, i responsabili decidono di sostituire DB2 con un altro RDBMS, per esempio

Oracle Server. A questo punto, l'unico sforzo da fare è far migrare i dati da DB2 ad Oracle, ma l'applicazione Java continuerà a funzionare come prima.

Questo vantaggio è un vantaggio determinante. Basti pensare alle banche o agli enti statali, che decine di anni fa si affidavano completamente al trittico Cobol-CICS-DB2 offerto da IBM. Adesso, con l'enorme mole di dati accumulati negli anni, hanno avuto difficoltà a migrare verso nuove piattaforme.

In futuro, con Java e JDBC, le migrazioni saranno molto meno costose. E già adesso è molto più semplice scrivere applicazioni che utilizzino diversi RDBMS contemporaneamente.

18.2 Le basi di JDBC

Come già asserito, JDBC è uno strato di astrazione software tra un'applicazione Java ed un database. La sua struttura a due livelli permette di accedere a RDBMS differenti, a patto che questi supportino l'ANSI SQL 2 standard. La stragrande maggioranza dei RDBMS in circolazione supporta come linguaggio di interrogazione un super-insieme dell'ANSI SQL 2. Ciò significa che esistono comandi che funzionano specificamente solo sui RDBMS su cui sono stati definiti (comandi proprietari) e che non sono parte dell'SQL standard. Questi comandi semplificano l'interazione tra l'utente e il database, sostituendosi a comandi SQL standard più complessi. Ciò implica che è sempre possibile sostituire ad un comando proprietario del RDBMS utilizzato un comando SQL standard, anche se l'implementazione potrebbe essere più complessa. Un'applicazione Java-JDBC che vuole mantenere una completa indipendenza dal RDBMS dovrebbe utilizzare solo comandi SQL standard, oppure prevedere appositi controlli laddove si vuole necessariamente adoperare un comando proprietario.

Il nome JDBC deriva dall'altra tecnologia di driver considerata standard: ODBC. ODBC è l'acronimo di Open Database Connectivity ed è un driver che tutti i database supportano di default.

I due fondamentali componenti di JDBC sono:

1. un'implementazione del vendor del RDBMS (o di terze parti) conforme alle specifiche delle API `java.sql`.
2. Un'implementazione da parte dello sviluppatore dell'applicazione.

18.2.1 Implementazione del vendor (Driver JDBC)

Il vendor deve fornire l'implementazione di una serie di interfacce definite dal package `java.sql`, ovvero `Driver`, `Connection`, `Statement`, `PreparedStatement`, `CallableStatement`, `ResultSet`, `DatabaseMetaData` e `ResultSetMetaData`. Ciò significa che saranno fornite alcune classi, magari impacchettate in un unico file archivio JAR, che implementano i metodi delle interfacce appena citate. Solitamente tali classi appartengono a package specifici ed i loro nomi sono spesso del tipo:

(per esempio: `DB2Driver`, `DB2Connection`...). Inoltre il vendor dovrebbe fornire allo sviluppatore anche una documentazione. Attualmente tutti i più importanti RDBMS supportano driver JDBC.

Esistono quattro tipologie diverse di driver JDBC caratterizzati da differenti potenzialità e strutture.

1. **Tipo 1: JDBC-ODBC Bridge Driver:** è un driver che ha un'interfaccia JDBC verso il programma Java, e un'interfaccia ODBC verso il database. Il driver riceve comandi dal programma Java con JDBC, e li passa al database tramite ODBC. La parola “bridge” infatti, in inglese significa “ponte”. È il driver dalle prestazioni peggiori. Sino alla versione 7 il JDK offriva un'implementazione di questo driver, con la versione 8 questa è stata rimossa.
2. **Tipo 2: Driver Native API Driver:** è un driver scritto in un linguaggio nativo come il C. Un esempio è il performante “OCI Driver” per il database Oracle. Essendo nativo, ha bisogno di essere installato, e la sua implementazione dipende dalla piattaforma.
3. **Tipo 3: Network Protocol Driver o Middleware Driver:** è un driver scritto in Java e quindi indipendente dalla piattaforma che viene installato non sulla macchina dove gira il programma Java, ma su una macchina remota. Questo ha la caratteristica di avere la responsabilità di interagire con il database (o i database) nella maniera più appropriata. Il programmatore si interfaccia col driver, e il driver fa da middleware verso uno o più database.
4. **Tipo 4: Pure Java Driver:** è un driver scritto in Java e quindi indipendente dalla piattaforma. Risiede sulla stessa macchina del client e quindi viene distribuito con il client stesso. Un esempio è il famoso “Thin Client” di Oracle.

Per gli esempi in questo modulo useremo come database “Java DB”, che è stato scaricato insieme al JDK. Per le note di installazione e la configurazione per far girare i programmi potete consultare l'appendice S on line.

18.2.2 Implementazione dello sviluppatore (Applicazione JDBC)

Lo sviluppatore ha un compito piuttosto semplice: implementare codice che sfrutta l'implementazione del vendor, seguendo pochi semplici passi.

Un'applicazione JDBC deve:

1. caricare un driver.
2. Aprire una connessione con il database.
3. Creare un oggetto `Statement` per interrogare il database.
4. Interagire con il database.
5. Gestire i risultati ottenuti.

Viene presentata di seguito una semplice applicazione che interroga un database, utilizzando come driver l'implementazione della Sun del bridge JDBC-ODBC, presente nella libreria standard di Java

(JDK1.1 in poi).

```
0 import java.sql.*;
1
2 public class JDBCApp {
3 public static void main (String args[]) {
4     try {
5         // Carichiamo un driver per connetterci a Java DB
6         String driver = "org.apache.derby.jdbc.EmbeddedDriver";
7         Class.forName(driver);
8         // Creiamo la stringa di connessione
9         String url = "jdbc:derby:Music";
10        // Ottieniamo una connessione con username e password
11        Connection con =
12            DriverManager.getConnection(url,"myUserName","myPassword");
13        // Creiamo un oggetto Statement per interrogare il db
14        Statement cmd = con.createStatement();
15        // Eseguiamo una query e immagazziniamone i risultati
16        // in un oggetto ResultSet
17        String qry = "SELECT * FROM Album";
18        ResultSet res = cmd.executeQuery(qry);
19        // Stampiamone i risultati riga per riga
20        while (res.next()) {
21            System.out.printf("%s : %s (%s)\n", res.getString("Artista"),
22                res.getString("Titolo"), res.getInt("Anno"));
23        }
24        res.close();
25        cmd.close();
26        con.close();
27    } catch (SQLException e) {
28        e.printStackTrace();
29    } catch (ClassNotFoundException e) {
30        e.printStackTrace();
31    }
32 }
33 }
34 }
```

18.2.3 Analisi dell'esempio JDBCApp

La nostra applicazione è costituita da un'unica classe contenente il metodo `main()`, non perché sia la soluzione migliore, bensì per evidenziare la sequenzialità delle azioni da eseguire.

Alla riga 0 viene importato l'intero package `java.sql`. In questo modo è possibile utilizzare i reference relativi alle interfacce definite in esso. I reference, sfruttando il polimorfismo, saranno utilizzati per puntare ad oggetti istanziati dalle classi che implementano tali interfacce, ovvero le classi fornite dal vendor. In questo modo l'applicazione non utilizzerà mai il nome di una classe fornita dal vendor, rendendo in questo modo l'applicazione stessa indipendente dal database utilizzato. Infatti gli unici riferimenti esplicativi all'implementazione del vendor risiedono all'interno di stringhe, facilmente parametrizzabili in svariati modi (come vedremo nei prossimi paragrafi).

Alla riga 7 utilizziamo il metodo statico `forName()` della classe `Class` (cfr. modulo 11) per caricare in memoria l'implementazione del driver JDBC, il cui nome completo viene specificato

nella stringa argomento di tale metodo. A questo punto il driver è caricato in memoria e si auto-registra con il `DriverManager` grazie ad un inizializzatore statico, anche se questo processo è assolutamente trasparente allo sviluppatore.

Questo passo va fatto un'unica volta all'interno di un'applicazione e, come vedremo più avanti, è anche evitabile per questo particolare driver che supporta la versione 4.1.

Tra la riga 9 e la riga 12 viene aperta una connessione al database mediante la definizione di una stringa `url`, che deve essere disponibile nella documentazione fornita dal vendor (avente sempre una struttura del tipo `jdbc:subProtocol:subName` dove `jdbc` è una stringa fissa, `subProtocol` è un identificativo del driver e `subName` è un identificativo del database) e tramite la chiamata al metodo statico `getConnection()` sulla classe `DriverManager`.

Alla riga 14 viene creato un oggetto `Statement` che servirà da involucro per trasportare le eventuali interrogazioni o aggiornamenti al database.

Tra la riga 17 e la riga 18 viene formattata una query SQL in una stringa chiamata `qry`, eseguita sul database, e vengono immagazzinati i risultati all'interno di un oggetto `ResultSet`. Quest'ultimo corrisponde ad una tabella formata da righe e colonne dove è possibile estrarre risultati facendo scorrere un puntatore tra le varie righe mediante il metodo `next()`.

Infatti, tra la riga 20 e 23, un ciclo `while` chiama ripetutamente il metodo `next()`, il quale restituirà `false` se non esiste una riga successiva a cui accedere. Quindi, finché ci sono righe da esplorare, vengono stampati a video i risultati presenti alle colonne di nome `Artista`, `Titolo` e `Anno`.

Tra la riga 29 e la riga 31 vengono chiusi il `ResultSet` `res`, lo `Statement` `cmd` e la `Connection` `con`.

Tra la riga 26 e la riga 27 viene gestita l'eccezione `SQLException`, sollevabile per qualsiasi problema relativo a JDBC, come una connessione non possibile o una query SQL errata.

Questa eccezione fornisce la possibilità di introdurre il codice di errore del vendor del database direttamente dentro la `SQLException`.

Tra la riga 26 e la riga 28 invece, viene gestita la `ClassNotFoundException` (sollevabile nel caso fallisca il caricamento del driver mediante il metodo `forName()`).

Il driver che stiamo usando per connetterci al database Java DB è di tipo 4.

Se volessimo eseguire la nostra applicazione per farla interagire con un particolare database, bisognerà cambiare alcuni dettagli. In particolare:

- riga 6: è possibile assegnare alla stringa `driver` il nome di un altro driver disponibile (opzionale);

- ❑ riga 9: è possibile assegnare alla stringa `url` il nome di un'altra stringa di connessione (dipende dal driver JDBC del database utilizzato e si legge dalla documentazione del driver);
- ❑ riga 12: è possibile sostituire le stringhe `myUserName` e `myPassword` rispettivamente con la `username` e la `password` per accedere alla fonte dei dati. Se non esistono `username` e `password` per la base di dati in questione, basterà utilizzare il metodo `DriverManager.getConnection(url)`.

18.3 Altre caratteristiche di JDBC

Esistono tante altre caratteristiche di JDBC. Nei prossimi paragrafi vedremo come gestire tutte le principali caratteristiche per interagire con i database. L'argomento in realtà è vasto, e gli esempi da fare potrebbero essere centinaia. Tuttavia ci limiteremo solo a ciò che è veramente essenziale per iniziare a programmare con Java Standard Edition.

18.3.1 Indipendenza dal database

Avevamo asserito che la caratteristica più importante di un programma JDBC è il poter cambiare il database da interrogare senza cambiare il codice dell'applicazione. Nell'esempio precedente però questa affermazione non trova riscontro. Infatti, se deve cambiare database, deve cambiare anche il nome del driver. Inoltre potrebbero cambiare anche la stringa di connessione (che solitamente contiene anche l'indirizzo IP della macchina dove è eseguito il database), lo `username` e la `password`. Come il lettore può notare però, queste quattro variabili non sono altro che stringhe, e una stringa è facilmente configurabile dall'esterno. Segue il codice dell'applicazione precedente, rivisto in modo tale da sfruttare un file di properties (cfr. paragrafo 13.1.1) per leggere le variabili che ci interessano:

```
import java.sql.*;
import java.util.*;
import java.io.*;

public class JDBCAppProperties {
    public static void main (String args[]) {
        Connection con = null;
        Statement cmd = null;
        ResultSet res = null;
        try {
            Properties p = new Properties();
            p.load(new FileInputStream("config.properties"));
            String driver = p.getProperty("jdbcDriver");
            Class.forName(driver);
            String url = p.getProperty("jdbcUrl");
            con = DriverManager.getConnection(url,
                p.getProperty("jdbcUsername"),
                p.getProperty("jdbcPassword"));
            cmd = con.createStatement ();
            String qry = "SELECT * FROM Album";
            ...
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        res = cmd.executeQuery(qry);
        while (res.next()) {
            System.out.printf("%s : %s (%s)\n", res.getString("Artista"),
                res.getString("Titolo"), res.getInt("Anno"));
        }
        res.close();
        cmd.close();
        con.close();
    } catch (SQLException | ClassNotFoundException | IOException e) {
        e.printStackTrace();
    }
}
}
}

```

18.3.2 Operazioni CRUD

Con JDBC è possibile eseguire qualsiasi tipo di comando CRUD (Create, Retrieve, Update, Delete) verso il database, non solo interrogazioni. Se volessimo inserire un nuovo record in una tabella potremmo scrivere:

```

String insertStatement = "INSERT INTO MyTable . . .";
int ris = cmd.executeUpdate(insertStatement);

```

Noi italiani siamo soliti chiamare “query” un qualsiasi comando inoltrato al database. In realtà in inglese il termine “query” (che si può tradurre come “interrogazione”) viene utilizzato solamente per i comandi di tipo `SELECT`. Tutti i comandi che in qualche modo aggiornano il database vengono detti “update”. Ecco perché con JDBC è necessario invocare il metodo `executeUpdate()` per le operazioni di `INSERT`, `UPDATE` e `DELETE`, e `executeQuery()` per le operazioni di `SELECT`.

Un aggiornamento del database non restituisce un `ResultSet`, ma solo un numero intero che specifica il numero di record aggiornati.

Per esempio, tenendo conto che la nostra unica tabella è stata creata dalla seguente istruzione SQL:

```

CREATE TABLE Album ( AlbumId int, Titolo varchar(20),
    Artista varchar(255), Anno int, PRIMARY KEY (AlbumId) );

```

che quindi ha quattro colonne, se volessimo inserire un nuovo record nel nostro database di esempio potremmo scrivere:

```

String insertStatement = String.format(
    "INSERT INTO ALBUM (AlbumId, Titolo, Artista, Anno) "
    + "VALUES (%s,'%s','%s',%s)", album.getAlbumId(), "
    + album.getArtista(), album.getTitolo(), album.getAnno());
stmt.executeUpdate(insertStatement);

```

poi potremmo aggiornarlo:

```
String updateStatement = String.format(  
    "UPDATE ALBUM set Titolo='%s', Artista='%s', Anno=%s "  
    + "WHERE AlbumId = %s", album.getArtista(), album.getTitolo(),  
    album.getAnno(), album.getAlbumId());  
stmt.executeUpdate(updateStatement);
```

e infine rimuoverlo:

```
String deleteStatement = String.format(  
    "DELETE FROM ALBUM WHERE AlbumId = %s", album.getAlbumId());  
stmt.executeUpdate(deleteStatement);
```

18.3.3 Statement parametrizzati

Esiste una sottointerfaccia di `Statement` chiamata `PreparedStatement`. Questa permette di parametrizzare gli statement ed è molto utile laddove esista un pezzo di codice che utilizza statement uguali, differenti solo per i parametri. Segue un esempio:

```
PreparedStatement stmt =  
    conn.prepareStatement("UPDATE Tabella3 SET m = ? WHERE x = ?");  
stmt.setString(1, "Hi");  
for (int i = 0; i < 10; i++) {  
    stmt.setInt(2, i);  
    int j = stmt.executeUpdate();  
    System.out.println(j + " righe aggiornate quando i=" + i);  
}
```

Un `PreparedStatement` si ottiene mediante la chiamata al metodo `prepareStatement()` inserendo dei punti interrogativi in luogo dei valori da parametrizzare. I metodi `setString()` (esistono anche i metodi `setInt()`, `setDate()` e così via) vengono usati per impostare i parametri. Si deve specificare come primo argomento un numero intero che individua la posizione del punto interrogativo all'interno del `PreparedStatement`, e come secondo argomento il valore che deve essere impostato.

Il metodo `executeUpdate()` (o eventualmente il metodo `executeQuery()`) in questo caso non ha bisogno di specificare query.

18.3.4 Stored procedure

JDBC offre anche il supporto alle stored procedure, mediante la sotto-interfaccia `CallableStatement` di `PreparedStatement`. Segue un esempio:

```
String spettacolo = "JCS";  
CallableStatement query = msqlConn.prepareCall(  
    "{call return_biglietti[?, ?, ?]}");  
try {
```

```

        query.setString(1, spettacolo);
        query.registerOutParameter(2,java.sql.Types.INTEGER);
        query.registerOutParameter(3,java.sql.Types.INTEGER);
        query.execute();
        int bigliettiSala = query.getInt(2);
        int bigliettiPlatea = query.getInt(3);
    } catch (SQLException SQLEX) {
        System.out.println("Query fallita");
        SQLEX.printStackTrace();
    }
}

```

In pratica, nel caso delle “stored procedure”, i parametri potrebbero essere sia di input sia di output. Nel caso siano di output, essi vengono registrati con il metodo `registerOutParameter()` specificando la posizione nella query e il tipo SQL.

18.3.5 Mappatura dei tipi Java - SQL

Esistono tabelle da tener presente per sapere come mappare i tipi Java con i corrispettivi tipi SQL. La tabella seguente mappa i tipi Java con i tipi SQL:

Tipo SQL	Tipo Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

La prossima tabella invece serve per avere sempre ben presente cosa restituiscono i metodi `getXXX()` di `ResultSet`:

Metodo	Tipo Java ritornato
getASCIIStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBoolean	boolean

getByte	byte
getBytes	byte[]
getDate	java.sql.Date
getDouble	double
getFloat	float
getInt	int
getLong	long
getObject	Object
getShort	short
getString	java.lang.String
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp

Infine è utile tener presente anche la seguente tabella che mostra quali tipi SQL sono associati ai metodi `setXXX()` di Statement:

Metodo	Tipo SQL del parametro del metodo
<code>setASCIIStream</code>	LONGVARCHAR prodotto da un ASCII stream
<code>setBigDecimal</code>	NUMERIC
<code>setBinaryStream</code>	LONGVARBINARY
<code>setBoolean</code>	BIT
<code>setByte</code>	TINYINT
<code>setBytes</code>	VARBINARY o LONGVARBINARY (dipende dai limiti relativi del VARBINARY)
<code>setDate</code>	DATE
<code>setDouble</code>	DOUBLE
<code>setFloat</code>	FLOAT
<code>.setInt</code>	INTEGER
<code>setLong</code>	BIGINT
<code>setNull</code>	NULL
<code>setObject</code>	L'oggetto passato è convertito al tipo SQL corrispondente prima di essere mandato
<code>setShort</code>	SMALLINT
<code>setString</code>	VARCHAR o LONGVARCHAR (dipende dalla dimensione relativa ai limiti del driver sul VARCHAR)
<code> setTime</code>	TIME
<code>setTimestamp</code>	TIMESTAMP

18.3.6 Transazioni

JDBC supporta anche le transazioni. Per usufruirne bisogna prima disabilitare l'auto commit nel seguente modo:

```
connection.setAutoCommit(false);
```

In seguito è possibile utilizzare i metodi `commit()` e `rollback()` sull'oggetto `connection`. Per esempio:

```
try {
    . . .
```

```

cmd.executeUpdate(INSERT_STATEMENT);
. . .
cmd.executeUpdate(UPDATE_STATEMENT);
. . .
cmd.executeUpdate(DELETE_STATEMENT);
conn.commit();
} catch (SQLException sqle) {
    sqle.printStackTrace();
    try {
        conn.rollback();
    }
    catch (SQLException ex) {
        throw new MyException("Commit fallito "
            + "- Rollback fallito!", ex);
    }
    throw new MyException("Commit fallito "
        + "- Effettuato rollback", sqle);
}
finally {
    // chiusura connessione. . .
}

```

18.4 Evoluzione di JDBC

Da quando JDBC è diventato un punto cardine della tecnologia Java, ha subito continui miglioramenti. Bisogna tener presente che anche il package `javax.sql` (definito da Sun come “JDBC Optional Package API”) fa parte dell’interfaccia JDBC sin dalla versione 1.4 di Java. In particolare con il JDK 1.8, ha introdotto la versione 4.2 dell’interfaccia JDBC. Questo non significa che dobbiamo utilizzare per forza tutte le novità di JDBC da subito necessariamente. Infatti la versione 4.2 include tutte le altre versioni precedenti:

- JDBC 4.1
- JDBC 4.0
- JDBC 3.0
- JDBC 2.1 core API
- JDBC 2.0 Optional Package API
- JDBC 1.2 API
- JDBC 1.0 API

Si noti che JDBC 2.1 core API e JDBC 2.0 Optional Package API di solito sono definite insieme come JDBC 2.0 API.

Sostanzialmente sino ad ora abbiamo parlato della versione 1.0. È importante conoscere le versioni

di JDBC perché così possiamo conoscere le caratteristiche supportate da un certo driver solamente riferendoci al supporto della versione dichiarata. In particolare, le classi, le interfacce, i metodi, i campi, i costruttori e le eccezioni dei package `java.sql` e `javax.sql` sono documentate con un tag javadoc “since” che specifica la versione. In inglese “since” significa “da” nel senso “esiste dalla versione”. Possiamo sfruttare tale tag per capire a quale versione di JDBC l’elemento documentato appartiene, tenendo presente la seguente tabella.

Tag	Versione JDBC	Versione JDK
Since 1.8	JDBC 4.2	JDK 1.8
Since 1.7	JDBC 4.1	JDK 1.7
Since 1.6	JDBC 4.0	JDK 1.6
Since 1.4	JDBC 3.0	JDK 1.4
Since 1.2	JDBC 2.0	JDK 1.2

Molte caratteristiche sono opzionali e non è detto che un driver le debba supportare. Per non avere brutte sorprese, è bene quindi consultare preventivamente la documentazione del driver da utilizzare.

18.4.1 JDBC 2.0

Oramai praticamente tutti i vendor hanno creato driver che supportano JDBC 2.0. Si tratta di un’estensione migliorata di JDBC che consente tra l’altro di scorrere il `ResultSet` anche al contrario, o di ottenere una connessione mediante un oggetto di tipo `DataSource` (package `javax.sql`) in maniera molto performante, grazie a una “connection pool”.

Per “connection pool” intendiamo quella tecnica di ottimizzazione delle prestazioni che permette di ottenere istanze già pronte per l’uso di oggetti di tipo connessione. In particolare questa tecnica è fondamentale in ambienti enterprise dove bisogna servire contemporaneamente, in maniera multithreaded, diversi client che chiedono connessioni. Il concetto di “pool” è stato già introdotto nel paragrafo 14.6.3.3.

In particolare i `DataSource` sono lo standard da utilizzare per le applicazioni lato server in ambienti Java EE (Enterprise Edition) e, giusto per avere un’idea di come si utilizzano, riportiamo il seguente frammento di codice:

```
InitialContext context = new InitialContext();
DataSource ds = (DataSource)context.lookup("jdbc/myDataSource");
Connection connection = ds.getConnection();
```

In questo caso abbiamo sfruttato la tecnologia JNDI per ottenere un’istanza dell’oggetto `DataSource` allo scopo di sfruttare un’eventuale connection pool “offertoci” direttamente da un server Java EE.

Dopo avere ottenuto un oggetto `Connection`, i nostri passi per interagire con il database non cambiano.

18.4.2 JDBC 3.0

Oggiorno non è assolutamente raro utilizzare driver JDBC 3.0. Tra le novità introdotte da JDBC 3.0 ricordiamo la capacità di creare oggetti di tipo `ResultSet` aggiornabili. Questo significa che abbiamo la possibilità di ottenere risultati e poterli modificare al volo in modalità “connessa”. Per esempio, con le seguenti istruzioni:

```
Statement stmt = con.createStatement()  
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);  
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");
```

si potrà scorrere il `ResultSet` senza che vengano mostrati eventuali cambiamenti ai dati nel database (con cui il `ResultSet` rimane connesso) e contemporaneamente sarà possibile modificarne i dati al volo. Per esempio, con le seguenti righe di codice:

```
rs.absolute(3);  
rs.updateString("NAME", "Luca");  
rs.updateRow();
```

aggiorniamo nella terza riga del `ResultSet`, con il valore “Luca”, la colonna “NAME”.

Si noti come, quando è possibile scorrere un `ResultSet`, è possibile non solo navigare indietro e in avanti, ma anche spostarsi ad una determinata riga con il metodo `absolute()`.

Oppure con il seguente frammento di codice:

```
rs.moveToInsertRow();  
rs.updateString(1, "Graziella");  
rs.updateString("COGNOME", "Giannoni");  
rs.updateBoolean(3, true);  
rs.updateInt(4, 32);  
rs.insertRow();  
rs.moveToCurrentRow();
```

abbiamo inserito una nuova riga nel `ResultSet`.

È interessante anche studiare la sottointerfaccia di `ResultSet` `RowSet` (package `javax.sql`). In particolare, `RowSet` è a sua volta estesa da `CachedRowSet` (package `javax.sql.rowset`), un tipo di oggetto che lavora in maniera disconnessa dal db, ma che può anche sincronizzarsi con un’istruzione esplicita. Per esempio, con le seguenti istruzioni (`rs` è un oggetto di tipo `CachedRowSet`):

```
rs.updateString(2, "Buba");  
rs.updateInt(4, 300);  
rs.updateRow();  
rs.acceptChanges();
```

il CachedRowSet si sincronizza con il db.

18.4.3 JDBC 4.0, 4.1 e 4.2

Esistono interessanti novità nella versione 4.0. Per esempio ora non è più necessario caricare un driver mediante il metodo `Class.forName()`. Infatti, mediante il meccanismo detto di “Service Provider” o anche “Java Standard Extension Mechanism” (in italiano “meccanismo di estensione standard di Java”), è possibile rendere il metodo `DriverManager.getConnection()` responsabile di trovare il giusto driver tra quelli disponibili al runtime. Bisogna però fare in modo che il driver da caricare contenga un file “META-INF/services/java.sql.Driver”, con all’interno una riga contenente il fully qualified name della classe del driver (ovvero il nome completo del package). Se per esempio la classe del driver da caricare è `Driver` del package `my.sql`, una riga del file deve contenere la seguente istruzione:

```
my.sql.Driver
```

Questo è esattamente il caso del driver di Java DB:

```
org.apache.derby.jdbc.EmbeddedDriver.
```

Ma non dobbiamo preoccuparci di implementare noi il meccanismo dell’estensione descritto, è già pronto.

È possibile verificare quanto detto aprendo il file derby.jar che è posizionato nella cartella cartella_jdk\db\lib.

Questo significa che negli esempi precedenti possiamo fare a meno dell’istruzione `Class.forName()`, tuttavia è stato riportato perché si potrebbero usare anche altri driver che supportano JDBC con versione minore di 4.0.

La versione JDBC 4.1 introdotta con Java 7, si discosta poco dal precedente modello 4.0. L’unica caratteristica che segnaliamo riguarda la possibilità di utilizzare il costrutto “try with resources” con tutti gli oggetti che si dovrebbero chiudere. Nel seguente esempio è possibile vedere come questo cambi radicalmente l’utilizzo della libreria in termini di semplicità. Per esempio la seguente implementa tutti i controlli che in realtà dovremmo fare sempre quando usiamo JDBC:

```
import java.sql.*;
public class TryWithResources1 {
    public static void main(String args[]) {
        selectFromDB();
    }
    public static void selectFromDB() {
        Connection conn = null;
        Statement stmt = null;
        ResultSet res = null;
        try {
            conn = DriverManager.getConnection("jdbc:derby:Music/*",
                "sa", "");
            stmt = conn.createStatement();
            res = stmt.executeQuery("SELECT * FROM Music");
            while (res.next()) {
                System.out.println(res.getString("Title"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if (conn != null)
                try {
                    conn.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            if (stmt != null)
                try {
                    stmt.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            if (res != null)
                try {
                    res.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
        }
    }
}
```

```
    "username", "password" * /);
stmt = conn.createStatement();
res = stmt.executeQuery("SELECT * FROM Album");
while (res.next()) {
    System.out.printf("%s : %s (%s)\n",
        res.getString("Artista"), res.getString("Titolo"),
        res.getInt("Anno"));
}
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    if (res != null) {
        try {
            res.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    res = null;
    if (stmt != null) {
        try {
stmt.close();
    } catch (SQLException e) {
e.printStackTrace();
    }
}
stmt = null;
if (conn != null) {
    try {
conn.close();
    } catch (SQLException e) {
e.printStackTrace();
    }
    conn = null;
}
}
}
}
```

Da Java 7 in poi è possibile riscrivere la classe precedente nel modo seguente:

```
import java.sql.*;

public class TryWithResources1 {
    public void selectFromDB() {
        try(Connection conn =
            DriverManager.getConnection(url, username, password);
            Statement stmt = conn.createStatement();
            ResultSet rs =
```

```
        stmt.executeQuery("SELECT * FROM PERSONA ")); {  
    while (rs.next()) {  
        System.out.println(rs.getString(1));  
    }  
} catch (SQLException e) {  
    e.printStackTrace();  
}  
}  
}
```

Il vantaggio sembra evidente.

Anche per la versione JDBC 4.2 introdotta con Java 8 non c'è molto da segnalare. Le classi `Date`, `Time` e `Timestamp`, del package `java.sql`, ora sono state riviste e sono stati aggiunti metodi per la conversione nelle nuove classi di `java.time` come `LocalDate`, `LocalTime` e `LocalDateTime`.

Un altro cambiamento degno di nota riguarda la possibilità di fare aggiornamenti se il conteggio delle righe finali eccede il numero `Integer.MAX_VALUE`. In casi come questo bisogna usare il metodo `executeLargeUpdate()`, che è stato introdotto nell'interfaccia `Statement` con Java 8.

Riepilogo

Questo modulo è stato dedicato alla gestione dei dati con Java. In particolare, su database relazionali che sfruttano il linguaggio SQL. È inevitabile che gli sviluppatori, prima o poi, abbiano a che fare con questo linguaggio, e quindi si raccomanda al lettore inesperto quantomeno qualche lettura e qualche esercizio di base su questo argomento. Le risorse gratuite su SQL in Internet sono fortunatamente diffusissime.

In questo modulo abbiamo dapprima esplorato la struttura e le basi dell'**interfaccia JDBC**. Poi abbiamo introdotto con esempi alcune delle caratteristiche più importanti ed avanzate come le **stored procedure** (con i `CallableStatement`), gli **statement parametrizzati** (con i `PreparedStatement`) e la gestione delle **transazioni**. In ultimo, ma di notevole importanza, è stata illustrata l'**evoluzione delle versioni di JDBC**. Infatti, capendo la differenza delle varie specifiche potremmo destreggiarci meglio tra i driver esistenti.

Esercizi modulo 18

Esercizio 18.a) JDBC. Vero o Falso:

1. Connection è solo un'interfaccia.
 2. Un'applicazione JDBC è indipendente dal database solo se si parametrizzano le stringhe relative al driver, lo URL di connessione, lo username e la password.
 3. Se si inoltra ad un particolare database un comando non standard SQL 2, questo comando funzionerà solo su quel database. In questo modo si perde l'indipendenza dal database, a meno

di controlli o parametrizzazioni.

4. Per eliminare un record bisogna utilizzare il metodo `executeQuery()`.
5. Per aggiornare un record bisogna utilizzare il metodo `executeUpdate()`.
6. `CallableStatement` è una sottointerfaccia di `PreparedStatement`. `PreparedStatement` è una sottointerfaccia di `Statement`.
7. Per eseguire una stored procedure bisogna utilizzare il metodo `execute()`.
8. L'autocommit è impostato a `true` per default.
9. In ambienti enterprise dove si eseguono applicazioni in contesti Java EE, `DataSource` va utilizzato in luogo di `Driver`.
10. Per poter modificare il risultato di una query, bisogna utilizzare un oggetto `RowSet` in luogo di un `ResultSet`.

Soluzioni esercizi modulo 18

Esercizio 18.a) JDBC, Vero o Falso:

1. Vero.
2. Vero.
3. Vero.
4. Falso.
5. Vero.
6. Vero.
7. Vero.
8. Vero.
9. Vero.
10. Falso.

**Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
-----------	-----------	---------

Saper spiegare la struttura dell'interfaccia JDBC (unità 18.1, 18.2)	<input type="checkbox"/>
Saper scrivere codice che si connette a qualsiasi tipo di database (unità 18.2, 18.3)	<input type="checkbox"/>
Saper scrivere codice che aggiorna, interroga e gestisce i risultati qualsiasi sia il database in uso (unità 18.2, 18.3)	<input type="checkbox"/>
Saper spiegare le tipiche caratteristiche avanzate di JDBC, come stored procedure, statement parametrizzati e transazioni (unità 18.3)	<input type="checkbox"/>
Comprendere le differenze tra le varie versioni delle specifiche JDBC (unità 18.4)	<input type="checkbox"/>

Note:

Interfacce grafiche: introduzione a JavaFX

Obiettivi:

Al termine di questo capitolo il lettore dovrebbe essere in grado di:

- ✓ Saper elencare le principali caratteristiche della tecnologia JavaFX (unità 19.1).
- ✓ Saper elencare le caratteristiche di JavaFX (unità 19.2).
- ✓ Saper scrivere semplici programmi con JavaFX (unità 19.3).
- ✓ Saper gestire i principali Layout Pane per costruire GUI complesse (unità 19.4).
- ✓ Essere in grado di creare semplici interfacce grafiche con il linguaggio FXML importandole in programmi JavaFX (unità 19.4).
- ✓ Capire come associare file o istruzioni CSS in un programma JavaFX (unità 19.5).
- ✓ Capire come creare gestori di eventi con il modello a delega di JavaFX (unità 19.6).
- ✓ Saper definire proprietà JavaFX personalizzate e saper sfruttare la tecnica del binding (unità 19.7).
- ✓ Imparare ad usare componenti grafici avanzati ed effetti speciali (unità 19.8, 19.9).

JavaFX è una tecnologia basata sul linguaggio Java. Essenzialmente si usa il linguaggio con tutte le sue potenzialità (espressioni lambda comprese) con una nuova libreria e con nuove caratteristiche come le proprietà JavaFX e il binding. Sino ad ora avevamo a disposizione due librerie come AWT e Swing per creare interfacce grafiche, complete, potenti e piene di funzionalità. Quindi perché c'è bisogno di JavaFX? Lo scopriremo nei prossimi due paragrafi.

19.1 Storia delle GUI in Java

L'acronimo GUI sta per “Graphical User interface”, che in italiano si traduce “interfaccia grafica utente”. Quando nacque Java nel 1995 in pieno boom internet, i programmi con interfacce grafiche erano nella maggior parte dei casi “applicazioni stand-alone” (ovvero che venivano eseguite direttamente in locale) e comunicavano opzionalmente via rete con server remoti. All'epoca non esistevano standard definitivi per i componenti grafici come oggi, e le interfacce erano tutte diverse tra loro e spesso strapiene di pulsanti e campi di testo. Ciononostante all'epoca le interfacce grafiche risultavano molto più moderne delle tipiche interfacce a riga di comando che erano state protagoniste per decenni. Il successo iniziale di Java fu dovuto in gran parte proprio all'introduzione di una nuova generazione di GUI che potevano essere visualizzate direttamente all'interno di un browser web: le applet. Queste eliminavano il bisogno di aggiornare le interfacce con installazioni in loco, perché bastava scaricare la nuova versione dal web per poter essere aggiornati. Il motto “Write Once Run Everywhere”, ovvero “scrivi una volta ed eseguirà ovunque”, che pubblicizzava la caratteristica di

Java di essere un linguaggio multipiattaforma, fu la chiave del successo iniziale. Purtroppo le applet non mantenne la promessa. Infatti si basavano sulla libreria nota come AWT (acronimo di “Abstract Window Toolkit”), la quale sfruttava i componenti grafici nativi del sistema operativo dove l’applet era lanciata. Dato che questi componenti differivano molto tra sistemi operativi diversi, la differenza tra interfacce grafiche tra sistema e sistema poteva risultare notevole. In pratica Java tradiva la sua filosofia di essere multipiattaforma. Fu introdotta quindi una nuova libreria grafica più potente di AWT: il framework Swing. Swing a differenza di AWT, non si basa su componenti grafici nativi del sistema operativo, bensì i componenti sono ridisegnati da Java allo stesso modo su tutti i sistemi operativi. Swing non ebbe molto successo all’inizio perché era molto più pesante di AWT e richiedeva risorse di sistema più elevate. Swing, insieme alla gestione automatica della memoria di Java, fu anche causa dell’iniziale pre-concetto basato sull’equazione “Java uguale programma lento”. Ma negli anni, grazie al miglioramento degli hardware, il gap tra un’interfaccia scritta con Swing o con AWT diminuì fino a diventare irrilevante. Intanto le applet erano state declassate ad applicazioni di secondo livello pesanti e potenzialmente pericolose, mentre altre tecnologie come Flash, Javascript e più tardi HTML 5 conquistarono il web. Nel 2007 Sun Microsystems introdusse la prima versione di una nuova tecnologia: JavaFX. Lo scopo era proprio quello di rientrare sul mercato Web e cercare di contrastare il dominio di Flash. Le prime versioni di JavaFX erano basate su un nuovo linguaggio di scripting chiamato “JavaFX Script”. Questo linguaggio di scripting non era paragonabile ad un linguaggio pulito, lineare, tipizzato e orientato agli oggetti come Java, e i programmatore non mostraron molto interesse. Come punto di forza il linguaggio proponeva multimedialità (mai supportata degnamente nella libreria standard di Java) ed effetti speciali (infatti la pronuncia di JavaFX dovrebbe assomigliare molto a “Java Effects”), cosa a cui i programmatore Java non erano molto abituati. Infatti multimedialità ed effetti speciali sono sempre stati due dei punti deboli del linguaggio e le librerie Java Media Framework (JMF) e Java2D e Java3D, sono sempre state considerate librerie di seconda classe. Infatti ogni piattaforma ha modi di supportare la multimedialità molto differenti, mentre gli effetti speciali sono da sempre troppo pesanti per un linguaggio object oriented che gestisce la memoria automaticamente.

Oracle nel 2011 ha pubblicato la versione 2.0 di JavaFX, con la quale non c’è più bisogno di imparare un nuovo linguaggio per programmare. Dalla versione 7 update 6 del Java Development Kit, JavaFX è stato inglobato nel pacchetto di installazione. Oggi, con l’avvento di Java 8, anche JavaFX è stato direttamente promosso alla versione 8.

Intanto Flash sta scomparendo del tutto, spodestato da HTML 5. Eppure c’è ancora bisogno dei cosiddetti “Rich Client”, ovvero “client ricchi” nel senso di GUI complesse e pesanti, che non si limitano solo a far eseguire video on line, o scorrere fotografie con uno slideshow, ma vere e proprie applicazioni che accedono a file in locale, a database e così via. Queste applicazioni vengono scaricate ed aggiornate via web, e non bisogna installarle. Inoltre Java 8 può essere eseguito anche su processori ARM, e il futuro si gioca molto sulla programmazione embedded che avrà sempre più spesso bisogno di interfacce grafiche. Oracle ha quindi deciso di puntare forte su JavaFX, andando a terminare lo sviluppo di Swing. Purtroppo non è stato possibile andare a inglobare le parti utili di JavaFX in Swing, perché tecnicamente ci sarebbe stato bisogno di una reingegnerizzazione completa. In questo libro quindi è stato deciso di eliminare il modulo relativo alla programmazione Java con Swing-AWT a favore di JavaFX, di cui offriremo un’introduzione che consentirà al lettore di iniziare a lavorare con le GUI. Purtroppo ci limiteremo solo ad un’introduzione, per entrare nei dettagli ci

vorrebbe un altro libro! Intanto chi vuole può comunque approfondire Swing e AWT con la corposa appendice Q on line. All'interno di questo modulo faremo comunque qualche raffronto tra i componenti di Swing/AWT e quelli di JavaFX per facilitare la migrazione a chi conosce già il vecchio modo di creare interfacce grafiche. Quindi lo studio dell'appendice Q è comunque consigliata.

I componenti di JavaFX e di Swing sono comunque interoperabili. Questo permetterà una migrazione graduale dalle interfacce create con le vecchie librerie a questa nuova tecnologia.

19.2 Caratteristiche di JavaFX

Di seguito elenchiamo le principali caratteristiche di JavaFX:

- ❑ JavaFX fornisce essenzialmente una nuova libreria Java che aggiunge nuove caratteristiche al linguaggio.
- ❑ È possibile usare il linguaggio FXML per costruire interfacce grafiche e farlo interagire con codice di business scritto in Java.
- ❑ Permette di essere interoperabile con le vecchie librerie AWT/Swing, ed è quindi possibile far coesistere vecchi componenti con nuovi componenti.
- ❑ Supporta le istruzioni e i file CSS per gestire gli stili.
- ❑ Fornisce componenti di alto livello come media player e browser basati sulla tecnologia WebKitHTML.
- ❑ Supporta un motore 3D e una libreria con oggetti 3D pronti per l'uso.
- ❑ Supporta il disegno tramite la libreria Canvas API.
- ❑ Supporta la stampa tramite la libreria Printing API.
- ❑ Supporta il testo in vari formati (Rich Text) e caratteri basati sullo standard Unicode.
- ❑ Permette la visualizzazione dei suoi componenti su risoluzioni Hi-DPI (ovvero le nuove risoluzioni oltre l'HD).
- ❑ Supporta il multi-touch sui dispositivi adeguati.
- ❑ Ingloba un motore di accelerazione grafica (Prism) che può essere utilizzato su diverse schede grafiche e GPU.
- ❑ Permette l'installazione inglobando il Java Runtime in modo tale da non richiederne l'installazione al client.

19.3 Hello World

Prima di fare qualsiasi considerazione su questa nuova tecnologia, iniziamo subito a codificare il

primo programma con JavaFX, così come facemmo nel primo modulo con Java: visualizziamo la scritta “Hello World!” con JavaFX. Il risultato è mostrato nell’immagine 19.1.



Figura 19.1 - Hello World con JavaFX.

Il codice per eseguire il nostro programma “Hello World” è il seguente:

```
import javafx.application.*;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.text.*;
import javafx.stage.*;

public class HelloWorld extends Application {
    @Override
    public void start(Stage stage) {
        Label label = new Label("Hello World!");
        label.setFont(new Font("Book Antiqua", 120));
        stage.setScene(new Scene(label));
        stage.setTitle("HelloWorld with JavaFX");
        stage.sizeToScene();
        stage.show();
    }
}
```

19.3.1 Analisi di HelloWorld

Iniziamo subito a notare che con JavaFX la nostra classe estende la classe astratta `Application`, (appartenente al package `javafx.application`) e deve ridefinire il metodo `start()` che rappresenta quello che rappresentava il metodo `main()` nelle classiche applicazioni Java. Il metodo `start()` prende in input un oggetto di tipo `Stage` (package `javafx.stage`). L’oggetto `stage` rappresenta il “Top Level Container”, ovvero la finestra grafica dove sarà visualizzata la nostra applicazione. Questa potrebbe essere una classica finestra di sistema (quella che in AWT si chiama `Frame`) o un pannello rettangolare se l’applicazione fosse eseguita come applet (nell’appendice Q c’è un paragrafo dedicato alla creazione di applet). Nel codice del metodo `start()` istanziamo inizialmente un oggetto di tipo `Label` dal package `javafx.scene.control`. Come si può capire dalla traduzione in italiano si tratta di una semplice etichetta dove poter scrivere testo. Infatti passiamo al costruttore la stringa che vogliamo visualizzare. Subito dopo andiamo a impostare il font (ovvero il tipo di carattere) sulla label stessa. Istanziando in questo caso un oggetto `Font` (package `javafx.scene.text`) passando al suo costruttore il nome del font e la dimensione. Poi con un’unica

istruzione impostiamo nell'oggetto `stage` un'istanza di `Scene` (package `javafx.scene`) al cui costruttore passiamo l'oggetto `label`. La classe `Scene` è la classe che deve essere usata per visualizzare del contenuto. Inoltre una `Scene` deve per forza essere inserita all'interno di un oggetto `Stage`. Con l'istruzione:

```
stage.setScene(new Scene(label));
```

abbiamo quindi inserito una `Label` in un oggetto `Scene`, e quest'oggetto `Scene` all'interno dell'oggetto `Stage`. Nella parte finale del metodo abbiamo impostato il titolo all'oggetto `scene`, poi su di esso abbiamo chiamato il metodo `sizeToScene()` per garantire che l'oggetto `stage` fosse ridimensionato in modo tale da visualizzare correttamente tutto il suo contenuto (metodo equivalente al metodo `pack()` della classe `java.awt.Window`). Infine con il metodo `show()` abbiamo fatto visualizzare l'applicazione.

Il ciclo di vita di una classe che estende `Application` è definito anche dai metodi `init()`, che viene invocato prima del metodo `start()`, e `stop()`, che viene invocato automaticamente quando l'applicazione viene chiusa. Questo ricorda molto da vicino il ciclo di vita di un applet.

È possibile verificare facilmente il ciclo di vita inserendo i seguenti metodi nel programma precedente:

```
@Override  
public void init() {  
    System.out.println("Inizio JavaFX");  
}  
@Override  
public void stop() {  
    System.out.println("Fine JavaFX");  
}
```

Come si potrà immaginare, questi due metodi possono essere usati per scopi più interessanti rispetto all'esempio. Con `init()` potremmo inizializzare l'applicazione, per esempio caricando dei dati da un database, mentre con `stop()` potremmo salvare lo stato dell'applicazione in un file (o sempre in un database).

19.3.2 Esecuzione di un'applicazione JavaFX

Le applicazioni JavaFX sono eseguite come ogni altra applicazione Java. Ma esistono diversi modi per far partire un'interfaccia grafica oltre a quello di usare la riga di comando, un editor come EJE, o un IDE come Eclipse. Infatti un eventuale utente finale non disporrà di questi strumenti. Possiamo creare un file batch (con suffisso ".bat") dove possiamo scrivere i comandi che useremmo da riga di comando (come con EJE), ma non tutti gli utenti sono così pratici da saper far partire un programma con un file di batch. Altri modi alternativi sono: la creazione di un file JAR eseguibile, inglobare in un applet la nostra applicazione, distribuirla con una tecnologia come Java Web Start. Per i JAR eseguibili e le applet è possibile trovare dei paragrafi nell'appendice Q, per Java Web Start è possibile consultare la pagina ufficiale di Oracle: <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136112.html>.

19.4 Creazione di interfacce complesse con i Layout

Nell'esempio precedente la difficoltà era quasi nulla poiché l'interfaccia era statica e non aveva una struttura articolata. Nella realtà creeremo interfacce ben più complesse della precedente. Il primo aspetto di cui ci andremo ad occupare è la gestione del posizionamento sull'interfaccia grafica dei componenti che vogliamo utilizzare. Infatti i componenti grafici di JavaFX si possono posizionare uno sull'altro per formare una vera interfaccia grafica. Per esempio su un oggetto `Scene` possiamo posizionare un oggetto `Stage`. E su di esso un pannello con una etichetta, un campo di testo e un altro pannello nella parte inferiore, che allinea un paio di pulsanti. Insomma le GUI sono fatte a strati. E gli oggetti contenitori decidono il posizionamento dei componenti che sono aggiunti su di essi. In generale un componente ha un componente padre su cui è posizionato, e zero o più componenti figli posizionati su di esso. Quindi è molto importante capire come si gestisce il posizionamento dei componenti grafici.

Con JavaFX ci sono essenzialmente due modi diversi per farlo:

1. programmaticamente con JavaFX (usando i layout manager come con Swing e AWT);
2. usando il linguaggio dichiarativo FXML per gestire il layout. È possibile creare file FXML anche utilizzando un tool grafico come Scene Builder.

19.4.1 I Layout pane di JavaFX

Il discorso dei layout sarà familiare a chi ha già programmato interfacce grafiche con Java. Anche in JavaFX è implementata la filosofia dei layout tramite i cosiddetti “layout pane”, ovvero dei pannelli che hanno un modo predefinito per posizionare i componenti che gli si aggiungono. Tutti i layout pane appartengono al package `javafx.scene.layout`.

19.4.1.1 Le classi `VBox` e `HBox`

I layout pane `VBox` e `HBox` rappresentano box verticali e orizzontali, e servono per arrangiare i componenti su di esso rispettivamente su una singola riga o su una singola colonna. Per esempio il seguente codice:

```
stage.setTitle("Motore di ricerca");
Button cercaButton = new Button("Cerca");
Button msfButton = new Button("Mi sento fortunato");
Label label = new Label("Inserisci parola da ricercare");
TextField text = new TextField();
VBox vBox = new VBox(GAP);
HBox northPane = new HBox(GAP);
HBox southPane = new HBox(GAP);
northPane.getChildren().addAll(label, text);
southPane.getChildren().addAll(cercaButton, msfButton);
southPane.setAlignment(Pos.CENTER);
vBox.getChildren().addAll(northPane, southPane);
northPane.setPadding(new Insets(GAP));
```

```
southPane.setPadding(new Insets(GAP));
stage.setScene(new Scene(vBox));
stage.show();
```

Mostra come comporre i due pannelli per organizzare una interfaccia grafica per un semplice motore di ricerca come mostrato in Figura 19.2.

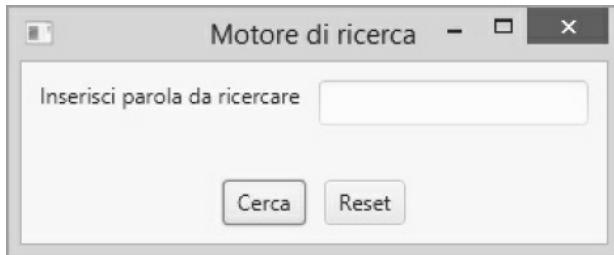


Figura 19.2 - VBox e HBox.

Il codice che istanzia i pulsanti, il campo di testo e la label, è molto intuitivo. La parte interessante riguarda il modo in cui abbiamo posizionato i componenti. Abbiamo istanziato un `VBox` (oggetto `vBox`) e due `HBox` (oggetti `northPane` e `southPane`). Poi abbiamo aggiunto tramite i metodi `addAll()` i componenti ai due `HBox`. E gli stessi `HBox` a `vBox`, che così ha allineato verticalmente i due box orizzontali.

La costante GAP usata nell'esempio precedente e passata in input agli oggetti `HBox`, `VBox` e `Insets`, è un double calcolato a partire dall'altezza del font di default. In teoria dovrebbe rappresentare il numero di pixel, ma visto che con i monitor di oggi la grandezza di un pixel è relativa, con questa tecnica si ha una spaziatura più regolare.

È importante sottolineare che dovremmo creare dei componenti riutilizzabili per poter programmare in modo più efficace. Per esempio il codice di prima che era contenuto interamente in un metodo `start()`, potrebbe essere riscritto per creare un componente pannello che estende `VBox` come segue:

```
//import omessi
public class MotoreDiRicercaPane extends VBox {
    private final static double gap =
        (0.8 * Font.getDefault().getSize());
    private Button cercaButton;
    private Button msfButton ;
    private Label label;
    private TextField text;

    public MotoreDiRicercaPane() {
        super(gap);
        cercaButton = new Button("Cerca");
        msfButton = new Button("Reset");
        label = new Label("Inserisci parola da ricercare");
        text = new TextField();
    }
}
```

```

        msfButton = new Button("Mi sento fortunato");
        label = new Label("Inserisci parola da ricercare");
        text = new TextField();
        setup();
    }

    public void setup() {
        HBox northPane = new HBox(gap);
        HBox southPane = new HBox(gap);
        northPane.getChildren().addAll(label, text);
        southPane.getChildren().addAll(cercaButton, msfButton);
        southPane.setAlignment(Pos.CENTER);
        getChildren().addAll(northPane, southPane);
        northPane.setPadding(new Insets(gap));
        southPane.setPadding(new Insets(gap));
    }
}

```

per poi potere essere utilizzato da una classe che ridefinisce il metodo `start()`:

```

//import omessi
public class MotoreDiRicercaStart extends Application {
    @Override
    public void start(Stage stage) {
        stage.setTitle("Motore di ricerca");
        MotoreDiRicercaPane vBox = new MotoreDiRicercaPane();
        stage.setScene(new Scene(vBox));
        stage.show();
    }
}

```

19.4.1.2 La classe BorderPane

Altro importante layout pane è `BorderPane`, che è l'equivalente di un pannello Swing/AWT a cui è associato un `BorderLayout`. È un pannello che dispone i suoi elementi solo in cinque posizioni: in alto, in basso, a destra, a sinistra e al centro. A differenza delle vecchie GUI, i componenti inseriti in queste aree non adatteranno le loro dimensioni per estendersi completamente in queste aree. Per esempio il seguente codice:

```

BorderPane pane= new BorderPane ();
pane.setTop(new Button("Alto"));
pane.setBottom(new Button("Basso"));
pane.setLeft(new Button("Sinistra"));
pane.setRight(new Button("Destra"));
pane.setCenter(new Button("Centro"));

```

produrrebbe l'interfaccia visualizzata in Figura 19.3.

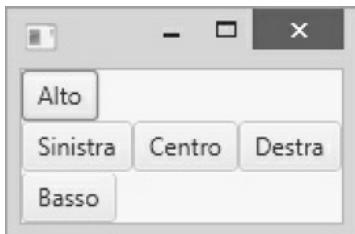


Figura 19.3 - BorderPane.

Volendo potremmo inserire il pannello del motore di ricerca all'interno di un componente che estende il BorderPane:

```
//import omessi
public class MotoreDiRicercaContainer extends BorderPane {
    private Label labelTop;
    private Label labelBottom;
    public MotoreDiRicercaContainer() {
        super();
        labelTop = new Label("Intestazione della pagina");
        labelBottom = new Label("Risultati della ricerca ");
        setup();
    }
    public void setup() {
        this.setTop(labelTop);
        this.setBottom(labelBottom);
        BorderPane.setAlignment(labelTop, Pos.CENTER);
        BorderPane.setAlignment(labelBottom, Pos.CENTER);
        this.setCenter(new MotoreDiRicercaPane());
    }
}
```

per poi utilizzarlo con una classe che estende Application ed ottenere l'interfaccia visualizzata in Figura 19.4.

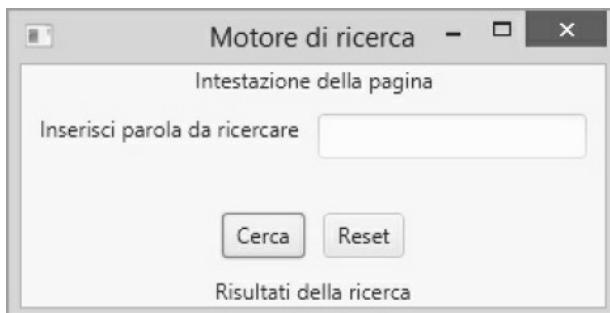


Figura 19.4 - BorderPane + componente motore di ricerca.

19.4.1.3 La classe GridPane

Il `GridPane` è un pannello dove è possibile posizionare i vari componenti nelle celle di una griglia immaginaria. Ogni componente però, può anche coprire l'area di più celle, sia in verticale che in orizzontale. In pratica è l'equivalente componente di un pannello con `GridBagLayout` di Swing/AWT, ma indubbiamente più semplice da utilizzare. Come esempio potremmo ricreare l'interfaccia del motore di ricerca fatta con `HBox` e `VBox` con il solo `GridPane`. Per esempio, potremmo creare un'estensione di `GridPane` con struttura del tutto simile a quella fatta per il componente di prima, ma con un nuovo metodo `setup()`:

```
public void setup() {  
    this.add(label,0,0);  
    this.add(text,1,0);  
    this.add(cercaButton,0,1);  
    this.add(msfButton,1,1);  
    GridPane.setAlignment(cercaButton, HPos.RIGHT);  
    GridPane.setAlignment(msfButton, HPos.LEFT);  
    this.setHgap(GAP);  
    this.setVgap(GAP);  
    this.setPadding(new Insets(GAP));  
}
```

Con questo codice posizioniamo gli elementi nella griglia come vogliamo, specificando il numero di riga e di colonna nel metodo `add()` come secondo e terzo parametro (gli indici partono da zero). In questo esempio abbiamo anche allineato orizzontalmente i due pulsanti per avvicinarli, tramite la chiamata al metodo statico `setAlignment()` (esiste anche il metodo `setValignment()`).

Se volessimo poi far espandere un componente per più di una cella è possibile specificare nel metodo `add()` anche il numero di righe e colonne che devono essere coperte dal componente:

```
this.add(labelBottom,0,2, 2,10);
```

dove abbiamo specificato che l'area dei risultati deve occupare cinque righe e una colonna. A scopo didattico aggiungendo questa istruzione:

```
this.setGridLinesVisible(true);
```

possiamo osservare i contorni delle celle della griglia. Le celle vuote avranno comunque uno spessore dovuto alle impostazioni del gap definito.

In Figura 19.5 abbiamo le due GUI affiancate, con e senza contorni.

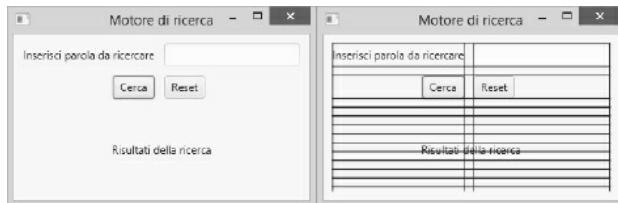


Figura 19.5 - Motore di ricerca GUI con GridPane.

19.4.1.4 Altri Layout Pane

Esistono anche altri layout pane che riassumiamo in questa tabella:

Classe	Descrizione
StackPane	Posiziona i componenti aggiunti su di esso uno sopra l'altro. Può essere usato per decorare i componenti per esempio con immagini.
TilePane	Posiziona i componenti aggiunti su di esso all'interno di una griglia dove tutte le celle hanno la stessa dimensione (come nel GridLayout di Swing/Awt).
FlowPane	Posiziona i componenti in un'unica riga, aggiungendo una nuova riga quando non c'è abbastanza spazio (come nel FlowLayout di Swing/Awt).
AnchorPane	Posiziona i componenti aggiunti su di esso in posizione assoluta, specificando le coordinate in pixel. Questo è il layout di default che usa attualmente Scene Builder.

19.4.1.5 FXML

FXML è un linguaggio dichiarativo basato su XML che consente di gestire le GUI in modo indipendente da Java. Questo tipo di gestione permette la separazione tra la parte statica di interfaccia e gli eventuali eventi da gestire su di essa (che vedremo nel prossimo paragrafo). È la stessa filosofia che si utilizza quando si programma per esempio per Android. Scrivere con un linguaggio di markup è possibile, ma spesso scomodo senza un tool adeguato. Un IDE come Netbeans 8, integra un FXML editor e consente anche l'integrazione con Scene Builder, un tool grafico per costruire le interfacce trascinando componenti grafici sui pannelli.

È anche possibile scaricare Scene Builder dalla stessa pagina dove abbiamo scaricato il JDK: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. Utilizzare un tool grafico dal nostro punto di vista è però limitante. Trascinare con il mouse widget grafici e posizionarli in dei pannelli, ridimensionarli e così via, ha il grande vantaggio di avere un'anteprima pressoché immediata di quello che si sta creando graficamente. Tuttavia il codice creato dal tool sarà meno utile.

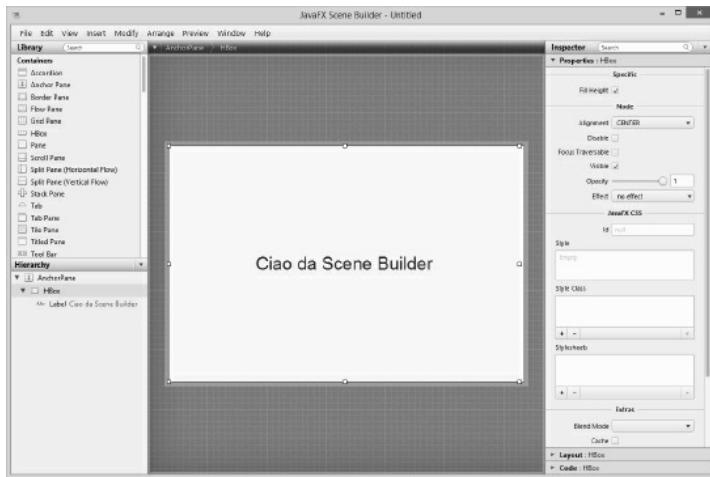


Figura 19.6 - Scene Builder in azione.

Probabilmente non organizzato nella maniera che vorremmo, e senza i controlli necessari affinché l’interfaccia rimanga consistente al variare delle desktop su cui gira il programma. Inoltre il tempo per posizionare un componente in un posto preciso, impostare le sue proprietà e così via, è molto spesso superiore a quello impiegato a scrivere direttamente il codice a mano. Riconosciamo però ai tool grafici l’utilità didattica, visto che generano codice senza che il programmatore lo conosca. Per tutte queste ragioni non mostreremo come funziona Scene Builder, ma ci limiteremo semplicemente a dare le informazioni essenziali per iniziare a programmare con FXML.

In questo esempio andiamo a realizzare una classica interfaccia di login con FXML:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.text.*?>
<?import javafx.scene.layout.*?>

<GridPane hgap="10" vgap="10">
    <padding>
        <Insets top="10" right="10" bottom="10" left="10"/>
    </padding>
    <children>
        <Label text="Username:" GridPane.columnIndex="0"
               GridPane.rowIndex="0" GridPane.halignment="RIGHT" />
        <Label text="Password: " GridPane.columnIndex="0"
               GridPane.rowIndex="1" GridPane.halignment="RIGHT" />
        <TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
        <PasswordField GridPane.columnIndex="1" GridPane.rowIndex="1" />
```

```
<HBox GridPane.columnIndex="0" GridPane.rowIndex="2"
      GridPane.columnSpan="2" alignment="CENTER" spacing="10">
    <children>
      <Button text="Login" />
      <Button text="Annulla" />
    </children>
  </HBox>
</children>
</GridPane>
```

Per prima cosa notiamo subito che, dopo la dichiarazione XML iniziale, sono utilizzati una serie di direttive per importare package da JavaFX. A questo punto viene dichiarato il tag `GridPane` con gap orizzontale e verticale impostato a 10 pixel. Inoltre come elemento interno a `GridPane`, viene impostato l'elemento `padding` che definisce un tag `Insets` esattamente equivalente all'impostazione che abbiamo visto con la programmazione JavaFX nell'esempio precedente del `GridPane`:

```
this.setPadding(new Insets(GAP));
```

Si noti anche come gli elementi innestati all'interno del tag `GridPane`, siano gli elementi che risiedono sul `GridPane`, e che quindi la logica di contenimento con FXML sia molto intuitiva. Il tag `children` e la relativa dichiarazione dei componenti grafici interni (riconoscibilissimi dopo gli esempi fatti precedentemente) equivale alla definizione dei metodi `add()` che abbiamo già visto. Sono specificati gli attributi `GridPane.columnIndex` e `GridPane.rowIndex`, che rappresentano rispettivamente l'indice di riga e l'indice di colonna, mentre `GridPane.halignment` consente di specificare l'allineamento orizzontale. Si noti che lo schema si ripete per il tag `HBox` che come `children` dichiara due pulsanti.

Dopo aver creato il file FXML, bisogna poi creare una classe che estende `Application`, la quale nella dichiarazione del metodo `start()` carica il file FXML e lo inserisce nell'oggetto `Stage`:

```
public void start(Stage stage) throws IOException {
  Parent root = FXMLLoader.load(getClass().getResource("Login.fxml"));
  stage.setScene(new Scene(root));
  stage.show();
}
```

Il risultato è visualizzabile in Figura 19.7.

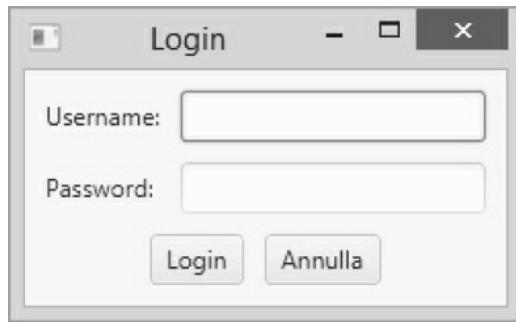


Figura 19.7 - Login con FXML.

Riguardo il pattern MVC (spiegato brevemente nell'appendice Q dedicata a Swing/ AWT e più tecnicamente su un datato articolo che potete scaricare insieme al codice all'indirizzo <http://www.claudiodesio.com/download/mvc.zip>) Oracle dichiara che è possibile implementarlo usando FXML per codificare il componente View, e il codice Java per implementare il componente Controller.

19.5 CSS

Il Cascading Style Sheets, meglio noto come CSS, è un linguaggio che definisce la formattazione dei documenti standard. Associare un file di stile ad un documento, significa poterlo sostituire (e quindi cambiare lo stile del documento) senza modificare il documento. Un'importante novità che viene definita in JavaFX è proprio la possibilità di utilizzare tale linguaggio nelle nostre interfacce. Aggiungere CSS ad un file FXML è molto semplice e naturale, infatti somiglia molto a quello che si fa con HTML. Per esempio potremmo creare un banale file CSS con una semplice istruzione di tipo "class", che rende le scritte delle label rosse, in grassetto e corsivo:

```
.label {  
    -fx-text-fill: rgb(255,0,0);  
    -fx-font-weight: bold;  
    -fx-font-style: italic;  
}
```

Si noti che è necessario utilizzare uno speciale insieme di attributi che iniziano per `-fx-`. I nomi degli attributi sono ricavati semplicemente dai nomi dei componenti grafici senza lettere maiuscole, e con i trattini separatori al posto della notazione "camel case". Per esempio `textAlignment` diventa `-fx-text-alignment`. Il reference per il CSS da utilizzare con JavaFX si può trovare all'indirizzo <http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>. Per caricare il file tramite FXML andiamo ad aggiungere l'attributo `stylesheet` all'elemento `GridPane` che contiene le label:

```
<GridPane hgap="10" vgap="10" stylesheets="login.css">
```

A questo punto la schermata di login precedente sarà visualizzata come in Figura 19.8.

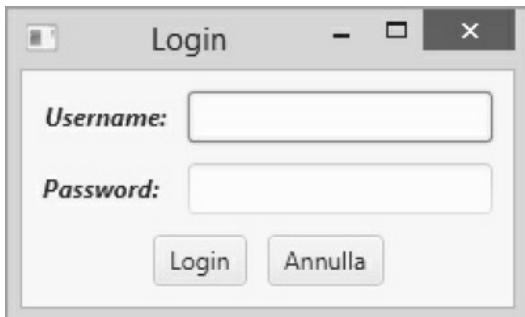


Figura 19.8 - Login con FXML e CSS.

Per applicare il nostro file CSS ad un file JavaFX è possibile usare invece questa sintassi, bisogna prima caricare il file CSS tramite l'oggetto `Scene`:

```
Scene scene = new Scene(pane);
scene.getStylesheets().add("test.css");
```

per poi caricare la classe `label` nel `GridPane` e rendere tutte le label personalizzate:

```
this.getStyleClass().add("label");
```

dove `this` è l'oggetto `GridPane`. Il risultato è visualizzabile in Figura 19.9.

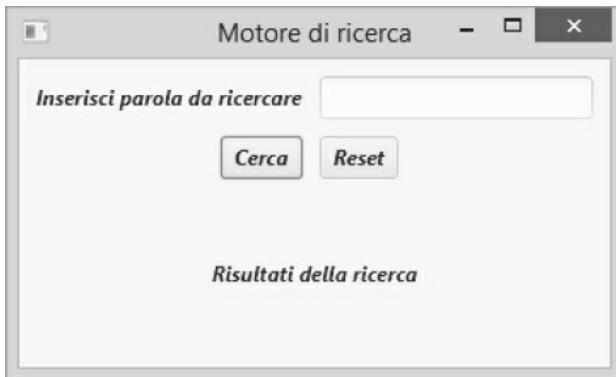


Figura 19.9 - Motore di ricerca con CSS.

È anche possibile creare nei file CSS (invece che nelle classi) stili personalizzati che poi vengono chiamati tramite un identificatore. Per esempio il seguente codice carica un'immagine di sfondo per i componenti che usano questo stile.

```
#back {
    -fx-background-image: url("rock.jpg");
```

}

Per far caricare l'immagine di sfondo al `GridPane`, è sufficiente la seguente istruzione:

```
this.setId("back");
```

Il pessimo risultato è visualizzato in Figura 19.10.

Sarebbe anche possibile applicare il CSS in maniera programmatica, anche se si perde il vantaggio di avere la separazione tra stile e codice Java. Per esempio per impostare l'immagine di sfondo anche ai pulsanti, basterà invocare il seguente metodo:

```
msfButton.setStyle("-fx-background-image: url(\"rock.jpg\")");
```



Figura 19.10 - Motore di ricerca con immagine di sfondo.

19.6 Gestione degli eventi

Per gestione degli eventi intendiamo la possibilità di associare l'esecuzione di una certa parte di codice in corrispondenza di un certo evento sulla GUI. Un esempio di evento potrebbe essere la pressione di un pulsante. In Java, che è sempre stato un linguaggio orientato agli oggetti, si è cercato di separare bene la logica degli eventi dalla logica di costruzione delle interfacce (nell'appendice Q c'è anche una parte che descrive l'evoluzione del modello ad eventi). Quindi in Java i "gestori degli eventi" sono oggetti separati che implementano interfacce come `ActionListener` o classi come `WindowListener`. Tramite il pattern noto come "Observer" (cfr. appendice Q) era possibile associare il gestore dell'evento alla sorgente dell'evento (per esempio un pulsante) e la virtual machine a quel punto si preoccupava di chiamare i metodi implementati nei nostri gestori degli eventi a seconda dell'evento che avveniva sulla sorgente. Era tutto perfetto, tranne il fatto che bisognava scrivere molto codice, e molto spesso la classe anonima risultava la scoria più veloce. Con l'avvento delle espressioni lambda sembra insensato utilizzare le classi anonime nella maggior parte dei casi. JavaFX dichiara altri tipi di interfacce da implementare, che sono appunto interfacce funzionali. Al posto della vecchia `ActionListener` ora esiste `javafx.event.EventHandler`, che

definisce un metodo `handle()`, la quale prende in input un oggetto di tipo `javafx.event.Event`. Quindi è tutto molto simile a quanto già esiste con Swing/AWT. Se volessimo associare ad un pulsante un'azione, per esempio reimpostare il campo di testo del motore di ricerca alla pressione del reset, allora potremmo scrivere:

```
resetButton.setOnAction(event -> text.setText(""));
```

19.7 Proprietà JavaFX e Bindings

In generale è sempre possibile programmare con questo tipo di gestione degli eventi e sfruttare tutte le possibilità attingendo dalla documentazione della libreria e dagli esempi on line. Tutto quello che si poteva fare prima con Swing/AWT, ora è possibile farlo con JavaFX, basta solo cercare un'equivalente interfaccia, metodo o classe adatta. Ma JavaFX consente anche di gestire gli eventi in maniera più avanzata. Infatti implementare il gestore degli eventi e il suo metodo SAM, che verrà chiamato quando viene premuto un pulsante, è semplice. La pressione del pulsante è astratta nella libreria come evento, e la JVM invoca il metodo SAM quando viene sollevato l'evento. Ma come sollevare un evento in base all'impostazione di una variabile d'istanza di un oggetto arbitrario? La tecnica si chiama “binding”, e consiste nell'associare le proprietà JavaFX di un oggetto all'esecuzione di metodi SAM in interfacce funzionali.

19.7.1 Proprietà JavaFX

Abbiamo definito la **proprietà** come un attributo di una classe che è possibile scrivere e leggere. In particolare esiste una convenzione che ci porta a definire le proprietà come variabili private con relativi metodi “accessor” e “mutator” (ovvero i cosiddetti metodi “set” e “get”). Questa convenzione deriva da una delle prime tecnologie Java oramai passata di moda, che tanto tempo fa era conosciuta come **JavaBeans**.

Il nome JavaBean però è ancora oggi sulla bocca di tutti, anche grazie al “riciclaggio” di tale termine nella tecnologia EJB (Enterprise Java Beans), ma non bisogna confondersi perché sono due tecnologie completamente differenti.

Questa tecnologia non si limitava a definire la “convenzione dei set e get”, bensì era proprio una specifica per classi. Le classi che rispettavano questa specifica venivano chiamate appunto **JavaBean** (ovvero chicco di caffè, che nella metafora di Sun erano i componenti dei nostri programmi che erano rappresentati da tazze di caffè) o più semplicemente **Bean**. Queste classi godevano della possibilità di risultare componenti riutilizzabili anche all'interno di tool grafici che tramite un semplice trascinamento ne definivano le gerarchie. Questi tool (tra cui ci piace ricordare l'antenato di Netbeans “Forte for Java”) permettevano (ma con molte limitazioni) azioni simili a quelle che oggi possiamo tranquillamente fare con strumenti quali Scene Builder. Tra le varie definizioni delle specifiche JavaBeans, esistevano (ed esistono tutt'ora) le cosiddette “bounded properties” (in italiano “proprietà limitate”), con le quali gli oggetti producevano eventi quando erano chiamati i metodi “set”.

Anche JavaFX dichiara delle proprietà, dette appunto **proprietà JavaFX**. Non si tratta però delle stesse specifiche di JavaBeans, ma è stato reimplementato un meccanismo molto simile per evitare al programmatore di scrivere tanto codice come si faceva con JavaBeans. Infatti le librerie JavaFX fanno tutto il “lavoro sporco” in background, e i programmatori possono concentrarsi su altri aspetti della programmazione. La differenza essenziale tra la specifica JavaBeans e le proprietà JavaFX consiste nel fatto che con le prime le proprietà coincidevano con le variabili d’istanza con metodi accessor e mutator, mentre in JavaFX esiste un terzo metodo che ritorna un oggetto che implementa l’interfaccia `Property`. In pratica è possibile associare un oggetto `Property` ad una variabile d’istanza. Per esempio se abbiamo una stringa `nome`, la convenzione di JavaFX ci porta a creare un metodo chiamato `nomeProperty()` che ritorna un oggetto `Property<String>`. Così, una eventuale classe `Utente` potrebbe essere dichiarata nel seguente modo:

```
import javafx.beans.property.*;
public class Utente {
    private StringProperty nome = new SimpleStringProperty("");
    public final StringProperty nomeProperty() {
        return nome;
    }
    public final void setNome(String nome) {
        this.nome.set(nome);
    }
    public final String getNome() {
        return nome.get();
    }
}
```

Si noti che `StringProperty` è una classe astratta, che è implementata da `SimpleStringProperty`. Oltre alla classe astratta `StringProperty` esistono anche `IntegerProperty`, `FloatProperty`, `LongProperty`, `DoubleProperty`, `BooleanProperty`, `ListProperty`, `SetProperty`, `MapProperty`, e per ogni altro tipo esiste `ObjectProperty<T>`. Poi è possibile attaccare un gestore di eventi alla proprietà. Quindi rispetto a JavaBeans è l’oggetto `Property` che scatena l’evento e non il bean.

Il **binding** (in italiano potremmo tradurlo come **legame**) è il concetto che porta due oggetti ad essere in una relazione tale che, se si aggiorna un oggetto deve essere aggiornato automaticamente anche l’altro. Più precisamente se viene aggiornata una proprietà di un oggetto deve essere aggiornata automaticamente la proprietà dell’altro oggetto. Ma per capire meglio le potenzialità del binding, possiamo creare per esempio due oggetti `Slider` legati con binding con un codice semplice come il seguente:

```
Slider slider1 = new Slider(0,250,500);
Slider slider2 = new Slider(0,250,500);
slider1.valueProperty().bindBidirectional(slider2.valueProperty());
```

il risultato sono le due slider mostrate in Figura 19.11.

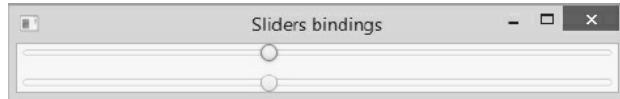


Figura 19.11 - Binding di due slider.

Muovendo una delle due slider, l'altra esegue lo stesso spostamento. Si noti che il metodo `bindDirectional()` è stato chiamato sull'oggetto di tipo `DoubleProperty` che ha ritornato il metodo `valueProperty()` della prima slider.

19.8 Effetti speciali

La denominazione di questa tecnologia è dovuta alla facilità con cui JavaFX consente di creare effetti speciali su componenti grafici come traslazioni, rotazioni, sfocature e così via. Per esempio con il seguente frammento di codice applichiamo un effetto di sfocatura (in inglese “blur”) alla `Label` con la scritta “Hello World!” del primo esempio che abbiamo fatto:

```
MotionBlur motionBlur = new MotionBlur();
motionBlur.setRadius(10);
motionBlur.setAngle(-90.0);
label.setEffect(motionBlur);
```

Abbiamo impostato come parametri per la sfocatura prima il raggio di 10 gradi e poi l'angolo di -90 gradi. In Figura 19.12 è possibile vederne l'effetto.



Figura 19.12 - Effetto blur.

Ottiene possiamo applicare delle transition, ovvero delle animazioni (package `javafx.animation`) come per esempio un'animazione nella quale la label sfuma (in inglese “fade”) e riappare:

```
FadeTransition fadeEffect = new FadeTransition(Duration.millis(3000));
fadeEffect.setFromValue(1.0);
fadeEffect.setToValue(0);
fadeEffect.setCycleCount(Animation.INDEFINITE);
fadeEffect.setAutoReverse(true);
fadeEffect.setNode(label);
fadeEffect.play();
```

Anche in questo caso il codice è molto intuitivo. Le chiamate ai metodi `setFromValue()` e

`setToValue()` definiscono quanto deve essere sfumato il componente. Il minimo valore possibile è 0 e il massimo è 1.0, questo significa che la scritta sfumerà sino a scomparire per poi riapparire. Si noti come per le animazioni sia la label ad essere passata all'animazione tramite il metodo `setNode()`. Si noti anche che l'animazione va fatta partire mediante il metodo `play()`. Un'altra animazione potrebbe essere quella della traslazione. Con il seguente frammento di codice andiamo a muovere un pulsante verso destra di 80 pixel per poi farlo tornare indietro:

```
TranslateTransition tt =  
    new TranslateTransition(Duration.millis(2000), button);  
tt.setByX(80f);  
tt.setCycleCount(Animation.INDEFINITE);  
tt.setAutoReverse(true);  
tt.play();
```

Il metodo `setByX()` definisce il verso della traslazione sull'asse x. Il metodo `setAutoReverse()` fa tornare indietro il componente al suo stato originale così come nel precedente esempio.

19.9 Componenti grafici avanzati

Come Sun creò per Swing l'applicazione dimostrativa SwingSet (ed altre) per mostrare le potenzialità del framework, con JavaFX Oracle ha creato le applicazioni dimostrative “JDK Demos & Samples” che potete scaricare all'indirizzo: <http://www.oracle.com/technetwork/java/javase/downloads/index.html> (alla fine della pagina). Conviene scompattare il file scaricato nella stessa cartella del JDK, e nella cartella scompattata troverete demo molto interessanti su JavaFX con codice incluso.

Un esempio semplice è quello che crea un grafico a torta come quello raffigurato in Figura 19.13. Il codice per generare un grafico come questo è molto semplice:

```
ObservableList<PieChart.Data> data =  
    FXCollections.observableArrayList(  
        new PieChart.Data("Jazz", 25),  
        new PieChart.Data("Blues", 24),  
        new PieChart.Data("Pop", 1),  
        new PieChart.Data("Classica", 15),  
        new PieChart.Data("Rock", 35));  
PieChart pieChart = new PieChart(data);  
pieChart.setTitle("Statistiche musica");
```

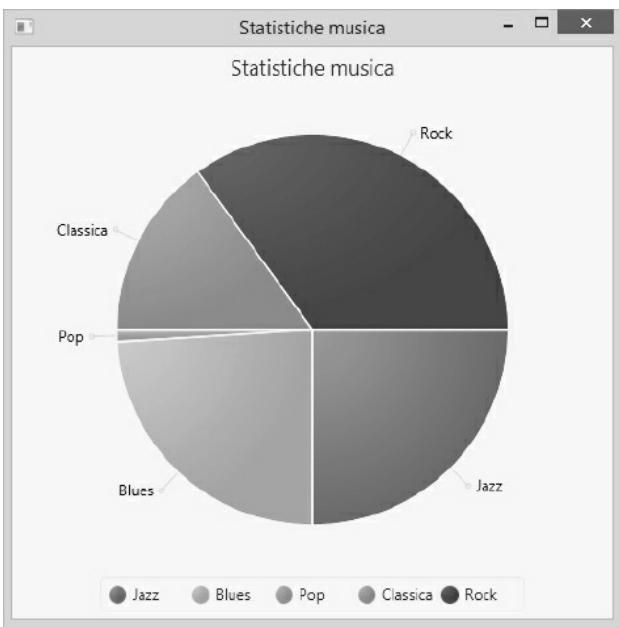


Figura 19.13 - Grafico a torta.

Come è possibile osservare viene creato una `ObservableList` di oggetti `PieChart.Data` con nomi e valori delle fette della torta. I dati vengono passati ad un oggetto `PieChart` a cui viene impostato il titolo. Il codice poi prosegue con il posizionamento del `PieChart` all'interno di un `box layout` e così via.

Uno dei punti deboli della libreria grafica di Java è sempre stata la parte multimediale. Il cosiddetto Java Media Framework in diciannove anni non è mai riuscito ad essere inglobato nella libreria standard di Java, vista la sua inefficienza. JavaFx invece, ha introdotto tra i componenti avanzati pronti da usare anche un media player (basato su “GStreamer multimedia framework”), dando finalmente una svolta anche al lato multimediale. Oggi per creare un video player ci vogliono pochissime righe di codice:

```
MediaPlayer mediaPlayer = new MediaPlayer(new Media(getPath()));
mediaPlayer.setAutoPlay(true);
MediaView view = new MediaView(mediaPlayer);
VBox box = new VBox(view);
```

Il metodo `getPath()` restituisce un percorso ad un file video mp4. Passiamo questo path ad un oggetto `Media` che a sua volta passiamo ad un oggetto `MediaPlayer`. Poi creiamo un oggetto `MediaView` a cui passiamo l'oggetto `mediaPlayer` per poi piazzarlo all'interno della nostra box. Il codice continua posizionando il box su un oggetto `Scene` che a sua volta viene piazzato su uno `Stage`. Qui non ci sono pulsanti per controllare il media player ma sarebbe un buon esercizio da fare! Un grosso aiuto lo trovate qui: <http://docs.oracle.com/javafx/2/media/playercontrol.htm>.

Un altro componente molto semplice da realizzare è un browser. In realtà era semplice da realizzare anche con Swing, ma le prestazioni sono nettamente migliorate con JavaFX. Infatti il componente WebView usa la tecnologia WebKitHTML, che consente di incorporare le pagine HTML nei componenti JavaFX. È possibile creare codice Javascript che chiama libreria JavaFX e viceversa.

Java 8 ha introdotto anche un nuovo e più performante motore Javascript di nome “Nashorn”.

Con le semplici seguenti righe di codice:

```
WebView browser = new WebView();
WebEngine webEngine = browser.getEngine();
webEngine.load("http://www.claudiodesio.com");
Scene scene = new Scene(browser);
```

otteniamo la creazione di un browser (banale e senza controlli) che visualizza il sito dell'autore e che vi raccomandiamo di visitare ogni tanto!



Figura 19.14 - Browser con <http://www.claudiodesio.com>.

Riepilogo

In questo modulo abbiamo introdotto la tecnologia che dovrebbe diventare la tecnologia standard per Java per creare interfacce grafiche utente, soppiantando definitivamente la coppia Swing/AWT. Esiste ancora interoperabilità tra le vecchie librerie e **JavaFX** e quindi sarà possibile migrare le vecchie interfacce in maniera meno drastica. Abbiamo iniziato questo modulo introducendo la storia delle GUI in Java per poi passare direttamente al primo approccio con il codice JavaFX, creando il primo programma "Hello World". Siamo poi passati a fare una breve panoramica sui layout pane principali di questa tecnologia, trovando grandi somiglianze con quelli definiti da AWT. Abbiamo in particolare esplorato e fatto degli esempi di utilizzo di `BorderPane`, `VBox`, `HBox` e `GridPane`. Poi

abbiamo visto come creare le nostre interfacce tramite il linguaggio dichiarativo **FXML**, separando così completamente la logica di controllo dalla logica di presentazione. Abbiamo anche visto come è possibile utilizzare i fogli di stile **CSS** per modificare lo stile delle nostre interfacce. Come la coppia Swing/AWT, la gestione degli eventi usa sempre il modello a delega ma com le interfacce rispetto ai classici “Listener” definiti in `java.awt`. Abbiamo anche notato quanto ci saranno utili le espressioni lambda, che possiamo utilizzare per specificare un gestore degli eventi al volo. Abbiamo anche esplorato le proprietà di JavaFX e la conseguente possibilità di utilizzare **bindings** tra componenti grafici (ma non solo). Infine abbiamo fatto dei semplici esempi dimostrativi su componenti grafici avanzati e toccato con mano qualche effetto speciale, visto che stiamo parlando comunque di una tecnologia chiamata JavaFX.

Esercizi modulo 19

Esercizio 19.a) GUI, AWT e Layout Manager, Vero o Falso:

1. È possibile creare interfacce miste che usano librerie di Swing e di JavaFX.
2. Per visualizzare un’interfaccia grafica bisogna utilizzare un oggetto `Stage` posizionato su un oggetto `Scene`.
3. Le interfacce create con JavaFX si creano usando layout pane, o quali sono pannelli che definiscono come i componenti aggiunti su di essi devono essere posizionati.
4. Il `GridPane` permette di posizionare i suoi componenti in una griglia le cui celle devono essere dello stessa dimensione.
5. Il linguaggio FXML permette di usare gli stessi componenti grafici che utilizza JavaFX in maniera dichiarativa.
6. È possibile usare un file CSS solo con interfacce create con FXML.
7. JavaFX diversamente da Swing/AWT, non gestisce gli eventi con un modello a delega. Infatti si usano le espressioni labda e non i gestori degli eventi.
8. Il binding permette a due oggetti che definiscono delle property JavaFX di legarsi e aggiornare il loro stato in base all’aggiornamento dell’altro oggetto.
9. Gli effetti speciali definiti JavaFX che permettono il movimento, terminano con il suffisso `Motion`, infatti estendono la classe astratta `Motion`.
10. È possibile eseguire un’applicazione JavaFX come applet.

Soluzioni esercizi modulo 19

Esercizio 19.a) GUI, AWT e Layout Manager, Vero o Falso:

- 1. Vero.**
- 2. Falso,** è esattamente il contrario.
- 3. Vero.**
- 4. Falso,** tali caratteristiche sono del `TilePane`.
- 5. Vero.**
- 6. Falso.**
- 7. Falso.**
- 8. Vero.**
- 9. Falso,** abbiamo visto un esempio di sfocatura tramite la classe `BlurMotion` che però non implementa movimenti ed estende la classe astratta `Effect`. Invece le classi che implementano un movimento estendono `Transition` e di conseguenza terminano con la parola `Transition`.
- 10. Vero.**

**Esercizi supplementari e altro materiale didattico sono disponibili on line agli indirizzi
<http://www.hoeplieditore.it/6291-1> e <http://www.claudiodesio.com/java8.html>.**

Obiettivi del modulo

Sono stati raggiunti i seguenti obiettivi?

Obiettivo	Raggiunto	In Data
Saper elencare le principali caratteristiche della tecnologia JavaFX (unità 19.1)	<input type="checkbox"/>	
Saper elencare le caratteristiche di JavaFX (unità 19.2)	<input type="checkbox"/>	
Saper scrivere semplici programmi con JavaFX (unità 19.3)	<input type="checkbox"/>	
Saper gestire i principali Layout Pane per costruire GUI complesse (unità 19.4)	<input type="checkbox"/>	
Essere in grado di creare semplici interfacce grafiche con il linguaggio FXML importandole in programmi JavaFX (unità 19.4)	<input type="checkbox"/>	
Capire come associare file o istruzioni CSS in un programma JavaFX (unità 19.5)	<input type="checkbox"/>	
Capire come creare gestori di eventi con il modello a delega di JavaFX (unità 19.6)	<input type="checkbox"/>	
Saper definire proprietà JavaFX personalizzate e saper sfruttare la tecnica del binding (unità 19.7)	<input type="checkbox"/>	
Imparare ad utilizzare componenti grafici avanzati ed effetti speciali (unità 19.8, 19.9)	<input type="checkbox"/>	

Note:

Simboli

! (NOT logico)
&& (short circuit AND)
& (AND bitwise o AND logico)
&= (AND e assegnazione)
<< (shift a sinistra)
<<= (shift a sinistra e assegnazione)
>> (shift a destra)
>>= (shift a destra e assegnazione)
>>> (shift a destra senza segno)
>>>= (Shift a destra senza segno e assegnazione)
@
@Deprecated (annotazione)
@Documented (annotazione)
@FunctionalInterface (annotazione)
@Inherited (annotazione)
@interface (definire annotazione)
@Native (annotazione)
@Override (annotazione)
@Repeatable (annotazione)
@Retention (annotazione)
@SuppressWarnings (annotazione)
@Target (annotazione)
^ (XOR bitwise)
^ (XOR logico)
| (OR bitwise o OR logico)
|= (OR e assegnazione)
|| (short circuit OR)
~ (NOT bitwise)
- (XOR e assegnazione)

A

abstract (parola chiave)

AnchorPane (classe)

Array

Arrays (classe)

assert (parola chiave)

Asserzioni

Astrazione

Attributo

Autoboxing e autounboxing

B

Binding

BlockingQueue (interfaccia)

boolean (parola chiave)

BorderPane (classe)

break (parola chiave)

Browser (implementare un)

byte (parola chiave)

Byte Stream

C

Cast di oggetti

automatico

catch (parola chiave)

char (parola chiave)

Character Stream

Ciclo foreach

Class (classe)

Classe

anonima
astratta
innestata
optional
wrapper
`Collection` (interfaccia)
`Collections` (classe)
Collezioni
 framework Collections
Collezioni eterogenee
`Comparable` (interfaccia)
`Comparator` (interfaccia)
Concorrenza
Consumer (interfaccia funzionale)
`continue` (parola chiave)
CSS

D

Date-Time API
 Codice legacy
Deadlock
Decorator (pattern)
Deduzione automatica del tipo
Deep copy (copia profonda)
`Deque` (interfaccia)
Diamond Problem
`do` (parola chiave)
dot (operatore)
`double` (parola chiave)
Duration (classe)

E

Eccezioni

checked e unchecked
personalizzate
`enum` (parola chiave)

Enumerazioni

`ChronoUnit`
`DayOfWeek`
innestate
membro
`Month`

Ereditarietà
multipla

Errori

Espressioni Lambda
composizione di

Espressioni regolari
classi predefinite
quantificatori

Executors

`extends` (parola chiave)

F

Factory Method

`File` (classe)

`Files` (classe)

`final` (parola chiave)

`finally` (parola chiave)

`float` (parola chiave)

`FlowPane` (classe)

`for` (parola chiave)

migliorato

Fork - Join

`Formatter` (classe)

`Function` (interfaccia funzionale)

Funzione anonima

G

Generici (Generics)

Geolocalizzazione

Gestione degli eventi

Grafico a torta

(implementare un)

GridPane (classe)

H

HBox (classe)

I

if (parola chiave)

import (parola chiave)

Incapsulamento

funzionale

osservazione

Inizializzatori

d'istanza

statici

Input da tastiera

InputStream (classe)

Instant (classe)

int (parola chiave)

Interfaccia

funzionale

Internazionalizzazione

Invarianti di classe

Invarianti interne

Invarianti sul flusso di esecuzione

Invocazione virtuale dei metodi

Istanziare

Iteratori

J

Java Compiler API

Java Database Connectivity (JDBC)

Java Development Kit (JDK)

Javadoc

JavaFX (GUI)

Java Runtime Environment (JRE)

JDBC. *See*

Java Database Connectivity (JDBC)

L

Lambda Calcolo

List (interfaccia)

lista dei parametri tipale, numerica e posizionale

Locale (classe)

Lock

long (parola chiave)

M

main (metodo)

Map (interfaccia)

Math (classe)

Media player (implementare un)

Membro

Metodo

astratto

bloccante con time out

bloccante indefinito

che lancia un'eccezione

che ritorna un booleano

che ritorna un elemento

costruttore

dichiarazione

di default

forEach()

generico

helper

notify()

notifyAll()

resume()

SAM

sleep()

start()

statico

suspend()

wait()

Modificatori

Monitor

Multithreading

N

Nashorn (motore Javascript)

NIO 2.0

O

Object (classe)

Oggetti Immutabili

Oggetto

Operatori

aritmetici

bitwise

d'assegnazione

di post-incremento (decremento)

di pre-incremento (decremento)

operatore ternario

priorità

relazionali

Operazioni CRUD
Operazioni di riduzione
`OutputStream` (classe)
Overload
Override

P

Package
 `java.lang`
 `java.time`
 `java.time.format`
 `java.time.temporal`
 `java.util`
 `java.util.concurrent.atomic`
 `java.util.concurrent.locks`
 `java.util.function`

Parametri
 covarianti
 passaggio di
 polimorfi

`Path` (interfaccia)
`Period` (classe)

Pipeline
Polimorfismo
 per dati
 per metodi

Postcondizione
Precondizione

`Predicate` (interfaccia funzionale)
Preemptive scheduling
Principio di inversione della dipendenza
`private` (parola chiave)
Progettazione per contratto
`protected` (parola chiave)
`public` (parola chiave)

Q

`Queue` (interfaccia)

R

Rappresentazione
binaria
decimale
esadecimale
ottale

`Reader` (classe)

Reference
a metodi

Reflection

Relazione “is a”

Riuso

`Runnable` (interfaccia)

S

Serializzazione di oggetti

`Set` (interfaccia)

Shallow copy (copia superficiale)

`short` (parola chiave)

Singleton (pattern)

`SortedMap` (interfaccia)

`SortedSet` (interfaccia)

`StackPane` (classe)

Statement parametrizzati

`static` (parola chiave)

blocco inizializzatore

Stored procedure

Stream

parallelisi

Stream API

String (classe)

StringTokenizer (classe)

super (parola chiave)

Supplier (interfaccia funzionale)

switch (parola chiave)

System (classe)

T

Temporal Adjusters

Temporal Queries

this (parola chiave)

Thread

classe

priorità

sincronizzazione

throw (parola chiave)

throws (parola chiave)

TilePane (classe)

Time slicing o round-robin scheduling

Tipi di dati

annotazioni

Generici

creare

mappatura dei tipi Java - SQL

non primitivi

primitivi

Tipi innestati

Transazioni su database

try (parola chiave)

Try with resources

Type erasure

U

`UnaryOperator` (interfaccia funzionale)

`Unicode`

`UTF`

V

`Varargs`

`Variabile`

- d'istanza

- locale

- statica

`VBox` (classe)

`Void` (classe)

`volatile` (parola chiave)

W

`Warnings`

`while` (parola chiave)

`Wildcard`

- bounded

- capture

`Writer` (classe)

X

`Xdoclet`

Informazioni sul Libro

La versione di Java più rivoluzionaria di sempre!

Il *Manuale di Java 8* è stato strutturato per soddisfare le aspettative di

- **Aspiranti programmati:** nulla è dato per scontato, è possibile imparare a programmare anche partendo da zero ed entrare nel mondo del lavoro dalla porta principale.
- **Studenti universitari:** le precedenti edizioni di quest'opera sono state adottate come libro di testo per molti corsi universitari nelle maggiori università italiane.
- **Programmatori esperti:** Java 8 è un linguaggio nuovo, molto diverso dal linguaggio che abbiamo usato sino ad ora. L'introduzione di decine di nuove caratteristiche, tra cui le espressioni lambda, obbligano chiunque ad aggiornarsi.

Circa l'autore

Claudio De Sio Cesari (<http://www.claudiodesio.com>) è consulente free lance specialista in formazione, sviluppo, analisi, progettazione, architettura object oriented. Ha collaborato con diverse Università, Enti Ministeriali ed aziende tra cui Sun Microsystems (ora Oracle) creatrice di Java, per cui è stato per diversi anni trainer e mentor.

