APPENDICI DEL MANUALE DI JAVA 9

(http://www.hoeplieditore.it/8302-2)

Indice generale

Appendice A

Breve storia di Java

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

✔ Conoscere per sommi capi la storia e l'importanza di Java (unità A.1, A.2, A.3, A.4).

Nei seguenti paragrafi sarà brevemente descritta la storia del linguaggio di programmazione Java. Negli anni essa è stata raccontata sui siti Sun e Oracle un po' come una leggenda, tanto che a volte ci è sembrato di leggere un romanzo storico, quasi mitologico. I protagonisti di questa storia però non sono eroi che solcano oceani e combattono singolari creature, ma personaggi come James Gosling, principale ideatore del linguaggio Java, Bill Joy, mitico vice-presidente di Sun Microsystems negli anni novanta (nonché artefice dell'editor VI, e padre del sistema operativo Solaris), e in generale del "Green Team", il gruppo di ingegneri (comprendenti Joy e Gosling) da cui sono nate tutte le idee. Non sapremo mai se gli aneddoti sono tutti veri, l'unica cosa certa è che oggi Java è il linguaggio di programmazione più usato al mondo.

A.1 Oak, Gosling e il Green Team

Java ha visto la luce a partire da ricerche in ambito universitario effettuate alla Stanford University all'inizio degli anni Novanta. Un piccolo team di ingegneri (poi soprannominato "Green Team") fu commissionato da Sun Microsystems per ricercare nuove soluzioni nel campo dei sistemi embedded, ovvero microsistemi di elaborazione con microprocessore creati per soddisfare ben determinati scopi. Esempi di sistemi embedded ben conosciuti ai nostri giorni sono smartphone, decoder, smart card e diversi elettrodomestici. L'effetto collaterale di queste ricerche

fu la creazione di un linguaggio progettato per programmare su questi sistemi, più semplice e meno pesante del linguaggio imperante in quegli anni: il C++.
Nel 1992 infatti, nacque il linguaggio Oak (in italiano "quercia"), il cui principale artefice fu James Gosling, oggi uno tra i più famosi e stimati "guru informatici" del pianeta. Il green team puntò alla realizzazione di un palmare sperimentale touch-screen che fu chiamato "*7" (ovvero "StarSeven") destinato all'home entertainment. Per diciotto mesi circa si chiusero in un anonimo ufficio in Sand Hill Road nella città di Menlo Park (Silicon Valley). Gosling creò Oak appositamente per programmare su questa demo, ma aveva ben in mente l'obiettivo di rendere il linguaggio indipendente dalla piattaforma dove i programmi dovevano girare.

Il nome Oak fu scelto perché fuori l'ufficio dove lavorò il Green Team si ergeva una quercia, a cui fu dedicato il linguaggio.

A.2 Java = Internet

In un primo momento Sun decise di sfruttare l'esperienza di *7 per entrare in campi che in quegli anni sembravano strategici come quello della domotica e quello della TV via cavo. Tuttavia i tempi non erano ancora maturi per argomenti quali il "video on demand" e gli "elettrodomestici intelligenti". Solo qualche anno dopo infatti, si è iniziato a richiedere video tramite Internet o TV e ancora oggi la domotica non è diffusa come potrebbe. Quindi fu accantonata l'idea originale per concentrarsi su un nuovo impiego di Oak, che si rivelò un linguaggio sorprendente.
Nel 1993, con l'esplosione di Internet (negli Stati Uniti), nacque l'idea di utilizzare codice eseguibile attraverso pagine HTML. La nascita delle applicazioni che utilizzavano la tecnologia CGI (Common Gateway Interface) rivoluzionò il World Wide Web in modo così rilevante da generare un incremento di utenti senza eguali nella storia dell'umanità.
Il mercato che sembrò più appetibile allora, divenne ben presto proprio Internet. Nel 1994 venne realizzato un browser che fu chiamato per breve tempo "Web Runner" (dal film "Blade Runner", il preferito di Gosling) e poi, in via definitiva, "HotJava". Non era un browser di primo livello, ma la sua creazione fece associare la parola "Java" alla parola "Internet".
Il 23 maggio del 1995 Oak, dopo un'importante rivisitazione, fu ribattezzato ufficialmente col nome "Java".

Java è il nome di una tipologia indonesiana di caffè molto famosa negli Stati Uniti, che pare fosse la preferita di Gosling.

Contemporaneamente la Netscape Corporation annunciava la scelta di dotare il suo allora celeberrimo browser, della Java Virtual Machine (JVM). Si trattava del software che permette di eseguire programmi scritti in Java direttamente all'interno del browser stesso, e conseguentemente su ogni piattaforma su cui girava Netscape Navigator. Questo significava una nuova rivoluzione nel mondo Internet: le pagine diventavano interattive a livello client grazie alla tecnologia "Applet". Gli utenti potevano per esempio utilizzare giochi direttamente sulle pagine web, ed usufruire di programmi quali chat dinamiche e interattive, il che rese il web molto più attraente.

A.3 Le ragioni del successo

In breve tempo inoltre, Sun Microsystems mise a disposizione il kit di sviluppo JDK (Java Development Kit) scaricabile gratuitamente dal sito http://java.sun.com. Nel giro di poche settimane i download del JDK 1.02a diventarono migliaia e Java iniziò ad essere sulla bocca di tutti. La maggior parte della pubblicità di cui usufruì Java nei primi tempi, era direttamente dipendente dalla possibilità di scrivere piccole applicazioni in rete, sicure, interattive ed indipendenti dalla piattaforma, chiamate **applet** (in italiano si potrebbe tradurre "applicazioncina"). Nei primi tempi sembrava che Java fosse il linguaggio giusto per creare siti Web spettacolari. Ma Java era molto di più che un semplice strumento per rendere più piacevole la navigazione. Inoltre ben presto furono realizzati strumenti per ottenere certi risultati con minor sforzo (vedi Flash di Macromedia). Tuttavia, la pubblicità che hanno fornito Netscape (che nel 1995 era un colosso che combatteva alla pari con Microsoft la "guerra dei browser") e successivamente altre grandi società quali IBM, Oracle, etc., ha dato i suoi frutti. Negli anni Java è diventato sempre di più la soluzione ideale a problemi che accomunano aziende operanti in settori diversi, per esempio banche, software house e compagnie d'assicurazione.

Con il nuovo millennio la tecnologia Java ha conquistato nuove nicchie di mercato come quello delle smart card, dei telefoni cellulari prima e degli smartphone poi (sistemi embedded per i quali era nato originariamente). Queste conquiste hanno consolidato il successo del linguaggio. Java ha infatti dato un grosso impulso alla diffusione di questi beni di consumo negli ul-

timi anni. Inoltre Java è leader nel mondo dello sviluppo web-enterprise. Infine grazie alla sua filosofia "write once, run everywhere" ("scritto una volta, può essere eseguito dappertutto"), la tecnologia Java è eseguibile su piattaforme completamente diverse tra loro. È persino potenzialmente eseguibile anche su congegni che non hanno ancora visto la luce oggi!
Nel 2006 Java è diventato open source (con la versione 6), e nel 2010 Sun Microsystems è stata rilevata da Oracle.
Oracle dopo un lungo periodo di adeguamento, ha iniziato ad evolvere la piattaforma Java. Nel 2011, la versione 7 di Java (la prima targata Oracle) non fu così stupefacente come ci si aspettava. Molte funzionalità previste non trovarono posto nella release, nonostante i numerosi ritardi nel tentativo di pubblicare tutto le novità che erano state promesse. La versione 8 (2014) invece ha rappresentato davvero un grande passo in avanti, ma anche questa subì diversi ritardi. Da settembre 2017, con il rilascio di Java 9, è stato adottato un nuovo modello di rilascio: ogni sei mesi viene rilasciata una nuova versione, contenente tutte le funzionalità pronte per la produzione, ritardando le funzionalità non pronte per le versioni future. Quindi, Java 10 è stato pubblicato nel marzo 2018, Java 11 nel settembre 2018 e non è difficile capire quali saranno le successive versioni e quando saranno rilasciate.

Per quanto riguarda le licenze e il supporto di Oracle, le cose sono cambiate, o meglio... stanno ancora cambiando. Java 11 è la prima versione LTS (Long Term Service) dopo Java 8 e, poiché lo scenario non è ancora chiaro, sarà meglio visitare questo link (in inglese) per avere informazioni più aggiornate: https://blog.joda.org/2018/08/java-is-still-available-at-zero-cost.html.

A.4 Java... run everywhere

Oggi Java è quindi un potente e affermatissimo linguaggio di programmazione e i suoi numeri sono impressionanti. Conta il più alto numero di sviluppatori attivi nel mondo (dodici milioni), oltre tre miliardi di device usano tecnologia Java, il 97% dei desktop aziendali, il 100% dei lettori blu-ray, un miliardo di download all'anno del runtime Java...
La tecnologia Java ha invaso la nostra vita quotidiana senza nemmeno ce ne rendessimo conto. È presente massicciamente, per esempio, nei nostri smartphone,

decoder, smart card, elettrodomestici, persino robot che passeggiano su Marte! Molto spesso Java è il motore invisibile degli aggeggi che usiamo ogni giorno, ed è probabilmente l'unico linguaggio di programmazione il cui nome è nel vocabolario anche di chi di programmazione non ne sa nulla. Con Java si programma per il sistema operativo mobile Google Android... la maggior parte delle applicazioni lato server fa uso di tecnologia Java... ed è il linguaggio più richiesto quando si cerca un lavoro nel campo informatico!

Appendice B

Preparazione dell'ambiente operativo su sistemi operativi Microsoft Windows: installazione del Java Development Kit

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

- ✔ Essere capace di installare il Java Development Kit correttamente su sistemi operativi Windows (unità B.1, B.2, B.3, B.4).

Se volete installare il JDK su Windows 10, potete anche far riferimento al mini video tutorial su YouTube che trovate all'indirizzo http://www.claudiodesio.com/yt/jdk.html. Altrimenti di seguito trovate i passi da fare per poter configurare il JDK su sistemi operativi Windows.

Non confondere il JDK (Java Development Kit) con il JRE (Java Runtime Environment). Il primo è l'ambiente che si usa per lo sviluppo Java (quello che ci serve), il secondo è solo l'ambiente di esecuzione.

B.1 Download JDK Standard Edition

Scaricare il Java Development Kit (versione più recente) da:

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

Il nome del file varia da un rilascio all'altro, ma non di molto. Nel momento in cui stiamo scrivendo la versione attuale è la 9.0.4, e il nome del file da scaricare per i sistemi a 64 bit è jdk-9.0.4_windows-x64_bin.exe. Il file è di circa 367 MB.

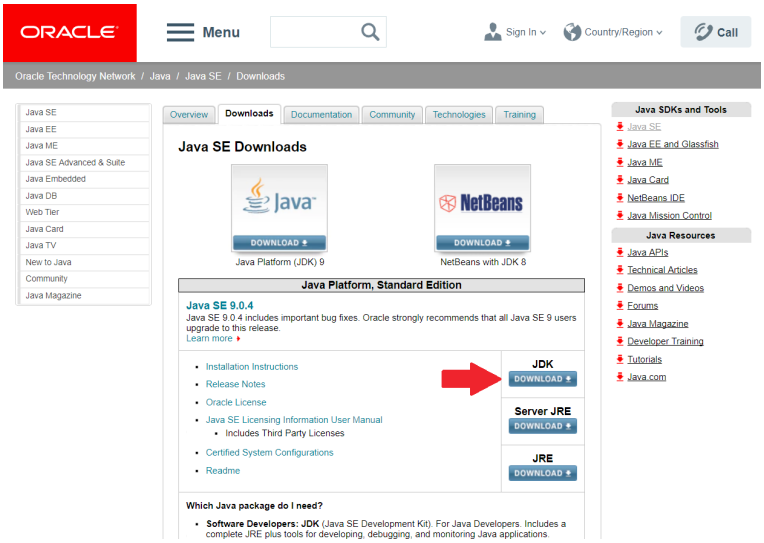


Figura B.1 - Schermata di download. La freccia evidenzia il giusto link su cui fare clic.

Per scaricare il file vi sarà chiesto di accettare una licenza (bisogna fare clic sull'opzione Accept License Agreement). Solo allora vi sarà permesso di scaricare il file corretto. Nella figura B.2 le frecce evidenziano i clic fondamentali del processo, da fare.

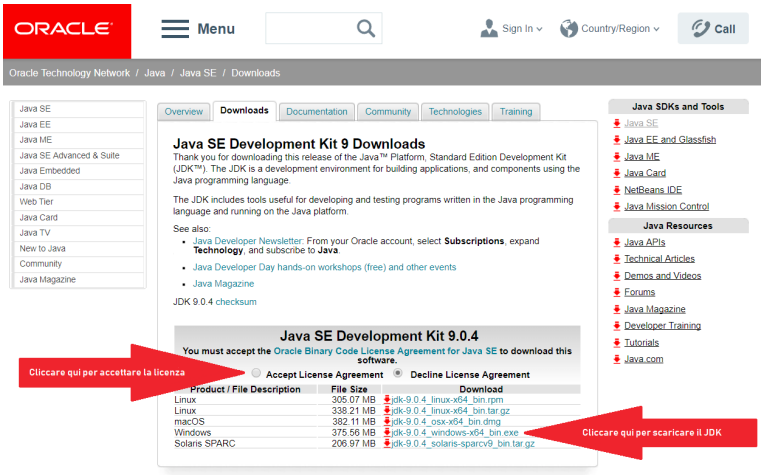


Figura B.2 - Schermata dove scaricare il JDK.

Una volta ottenuto il file di installazione, bisogna eseguirlo e seguire il procedimento guidato. Si tratta di un wizard molto semplice per il quale basterà avanzare di pagina in pagina senza scegliere opzioni diverse da quelle proposte di default. Sarà quindi installato il JDK.

Figura B.3 - Schermata di installazione del JDK.

Alla fine del processo verrà confermato il successo dell'installazione, ma non abbiamo ancora finito.

B.2 Download Documentazione API

Questo passo è teoricamente opzionale visto che le API sono sempre disponibili on line all'indirizzo http://docs.oracle.com/javase/9/docs/api. **Consigliamo comunque di avere la documentazione sempre a portata di mano, scaricata sul proprio coputer. Dalla versione 9 inoltre è disponibile la funzionalità di ricerca direttamente nell'interfaccia.**

Torniamo ora alla pagina:

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

Scaricare anche la documentazione "API Java". Fare clic sul pulsante download nella parte bassa della pagina sotto la voce Java SE 9 Documentation, come mostrato in figura B.4.

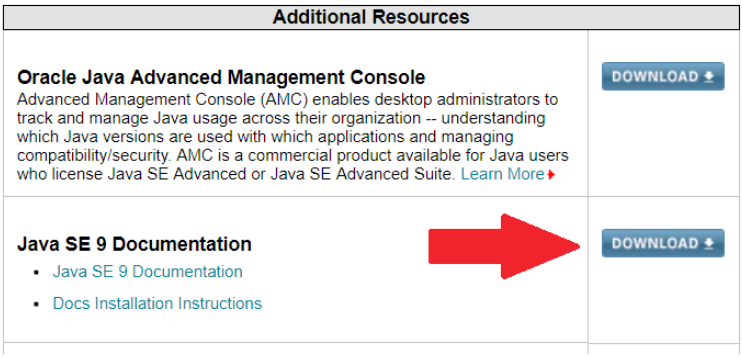


Figura B.4 - Schermata di Download, sezione "Additional Resources".

Verrete indirizzati ad un'altra pagina di download dove sarete in grado di scaricare (dopo aver accettato la licenza) la documentazione della libreria Java, come è possibile vedere in figura B.5.

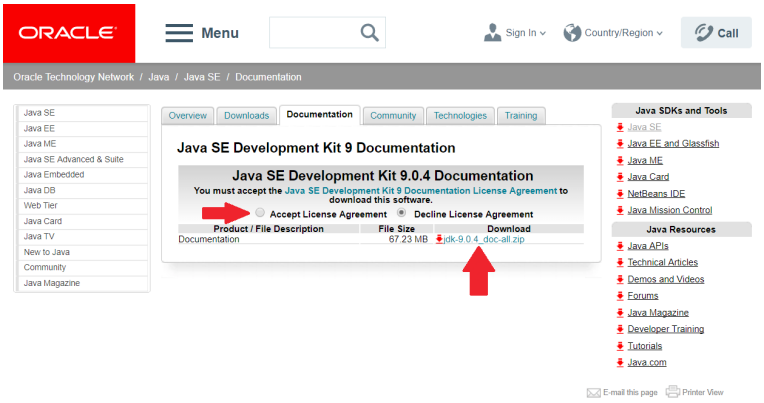


Figura B.5 - Schermata dove scaricare la documentazione.

Si tratta di un file .zip attualmente denominato jdk-9.0.1_doc-all.zip. Anche in questo caso il nome del file varia da un rilascio all'altro, ed il file è attualmente di circa 67 Mb. Decomprimere il file nella stessa cartella del JDK (o in una qualsiasi altra cartella che sia facilmente accessibile) nella cartella Docs (o scegliere un nome adeguato). Creare un collegamento sul desktop (o in qualsiasi altra posizione che si ritiene facilmente accessibile) al file index.html della cartella Docs.

B.3 Impostazione variabile PATH

Per poter utilizzare il compilatore (javac), l'interprete (java) e tutto il software presente nella suite di programmi contenuta nella cartella bin del JDK, bisogna impostare la variabile d'ambiente **PATH**, facendola puntare alla cartella bin del JDK.

B.3.1 Windows 10

Per i sistemi Windows 10 eseguire i seguenti passi:

1. aprire il pannello di controllo e cercare nella nuova finestra la frase Modifica le variabili d'ambiente relative al sistema;
2. nella nuova finestra che viene aperta, selezionare il tab Avanzate e fare clic su Variabili d'ambiente;
3. tra le Variabili di sistema (o se preferite tra le Variabili dell'utente) selezionare la variabile PATH e fare clic su Modifica;
4. sulla nuova finestra che viene presentata fare clic su Nuovo;
5. spostarsi nella casella Valore variabile e portarsi con il cursore all'inizio della riga, quindi aggiungere il percorso alla cartella bin del JDK, che dovrebbe essere simile a:

```
C:\Program Files\Java\jdk-9.0.4\bin
```

6. dopo aver confermato l'inserimento della nuova voce, selezionarla e portarla all'inizio della lista facendo clic sul pulsante Sposta su;
7. fare clic su OK e l'installazione è terminata.

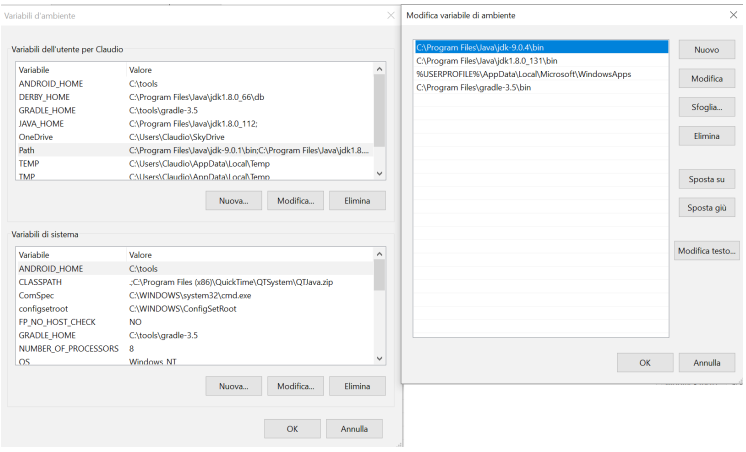


Figura B.6 - Impostazione variabile PATH su Windows 10.

B.3.2 Windows 8

Per i sistemi Windows 8, eseguire i seguenti passi:

1. aprire il pannello di controllo, e cercare nella nuova finestra la frase Modifica le variabili d'ambiente relative al sistema;
2. nella nuova finestra che viene aperta, selezionare Impostazioni di sistema avanzate e fare clic su Variabili d'ambiente;
3. tra le Variabili di sistema (o se preferite tra le Variabili utente) selezionare la variabile PATH e fare clic su Modifica;
4. spostarsi nella casella Valore variabile e portarsi con il cursore all'inizio della riga, quindi aggiungere il percorso alla cartella bin del JDK, che dovrebbe essere simile a:

```
C:\Program Files\Java\jdk-9.0.4\bin;
```

   il ";" ci permetterà di separare il nostro indirizzo dagli altri riferimenti;
5. fare clic su OK e l'installazione è terminata.

B.3.3 Windows 7

Per i sistemi Windows 7, eseguire i seguenti passi:

1. premere il tasto start, quindi il tasto destro su Risorse del Computer e fare clic su Proprietà;
2. selezionare Impostazioni di sistema avanzate e fare clic su Variabili d'ambiente;
3. tra le Variabili di sistema (o se preferite tra le Variabili utente) selezionare la variabile PATH e fare clic su Modifica;
4. spostarsi nella casella Valore variabile e portarsi con il cursore all'inizio della riga, quindi aggiungere il percorso alla cartella bin del JDK, che dovrebbe essere simile a:

```
C:\Program Files\Java\jdk-9.0.4\bin;
```

   il ";" ci permetterà di separare il nostro indirizzo dagli altri riferimenti;
5. fare clic su OK e l'installazione è terminata.

B.4 Verifica installazione

Per verificare che tutto sia andato a buon fine, aprite una prompt di comando e, nel campo di testo disponibile nel menu Start di Windows, digitate il comando:

```
cmd
```

Aperta la prompt dei comandi per testare la versione di Java installata, digitare:

```
java –version
```

se tutto è a posto, dovrebbe essere stampato un messaggio simile al seguente:

```
java version "9.0.4"
Java(TM) SE Runtime Environment (build 9.0.4+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.4+11, mixed mode)
```

Testare anche che il compilatore sia eseguito correttamente digitando il comando:

```
javac
```

se tutto è a posto, dovrebbe essere stampato un messaggio simile al seguente:

```
Usage: javac <options> <source files>
where possible options include:
  @<filename>                  Read options and filenames from file
  -Akey[=value]                 Options to pass to annotation processors
  --add-modules <module>(,<module>)*
        Root modules to resolve in addition to the initial modules,
        or all modules on the module path if <module> is ALL-MODULE-PATH.
  --boot-class-path <path>, -bootclasspath <path>
        Override location of bootstrap class files
  --class-path <path>, -classpath <path>, -cp <path>
        Specify where to find user class files and annotation
  processors
  -d <directory>               Specify where to place generated
  class files
  -deprecation
        Output source locations where deprecated APIs are used
  -encoding <encoding>         Specify character encoding used by
  source files
  -endorseddirs <dirs>         Override location of endorsed
  standards path
  -extdirs <dirs>              Override location of installed
  extensions
  -g                           Generate all debugging info
  -g:{lines,vars,source}       Generate only some debugging info
  -g:none                      Generate no debugging info
```

```
  -h <directory>
        Specify where to place generated native header files
  --help, -help                Print this help message
  --help-extra, -X             Print help on extra options
  -implicit:{none,class}
        Specify whether or not to generate class files for
  implicitly referenced files
  -J<flag>                    Pass <flag> directly to the runtime
  system
  --limit-modules <module>(,<module>)*
        Limit the universe of observable modules
  --module <module-name>, -m <module-name>
        Compile only the specified module, check timestamps
  --module-path <path>, -p <path>
        Specify where to find application modules
  --module-source-path <module-source-path>
        Specify where to find input source files for multiple modules
  --module-version <version>
        Specify version of modules that are being compiled
  -nowarn                      Generate no warnings
  -parameters
        Generate metadata for reflection on method parameters
  -proc:{none,only}
        Control whether annotation processing and/or compilation is done.
  -processor <class1>[,<class2>,<class3>...]
        Names of the annotation processors to run; bypasses default
  discovery process
  --processor-module-path <path>
        Specify a module path where to find annotation processors
  --processor-path <path>, -processorpath <path>
        Specify where to find annotation processors
  -profile <profile>
        Check that API used is available in the specified profile
  --release <release>
        Compile for a specific VM version. Supported targets: 6, 7, 8, 9
  -s <directory>               Specify where to place generated
  source files
  -source <release>
        Provide source compatibility with specified release
  --source-path <path>, -sourcepath <path>
        Specify where to find input source files
  --system <jdk>|none          Override location of system modules
  -target <release>            Generate class files for specific VM version
  --upgrade-module-path <path>
        Override location of upgradeable modules
  -verbose                     Output messages about what the
  compiler is doing
  --version, -version          Version information
  -Werror                      Terminate compilation if warnings occur
```

È possibile che sul vostro sistema sia già stato installato un altro JDK in preceden-

za (potreste anche non saperlo), quindi per essere sicuri che tutto sia stato fatto a dovere, conviene anche eseguire il seguente comando:

```
javac  -version
```

che dovrebbe stampare la versione attuale (nel nostro caso la 9.0.4):

```
javac 9.0.4
```

Appendice C

Comandi base per interagire con la riga di comando di Windows

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

✔ Essere capace di usare dei semplici comandi DOS per muoversi tra le cartelle di Windows (unità C).

Windows è un sistema operativo che è nato a partire da uno storico sistema operativo di nome **MS DOS**. Prima di diventare indipendente da DOS, i sistemi Windows altro non erano che un'estensione con interfaccia grafica di DOS. Era possibile aprire il sistema DOS a partire da Windows con una voce del menu Start chiamata **Prompt di DOS**. Da Windows 7 questa voce è sparita, e per aprire una sessione DOS (che ora viene definita anche **riga di comando** o **prompt dei comandi**) è possibile digitare l'istruzione:

```
cmd
```

nel campo di test visibile facendo clic sul pulsante Start, per Windows 8 semplicemente digitando l'istruzione nel campo di ricerca in alto a destra del desktop, mentre per Windows 10 nel campo di testo in basso a sinistra del desktop. Dovrebbe apparire una finestra simile a quella visualizzata in figura C.1.

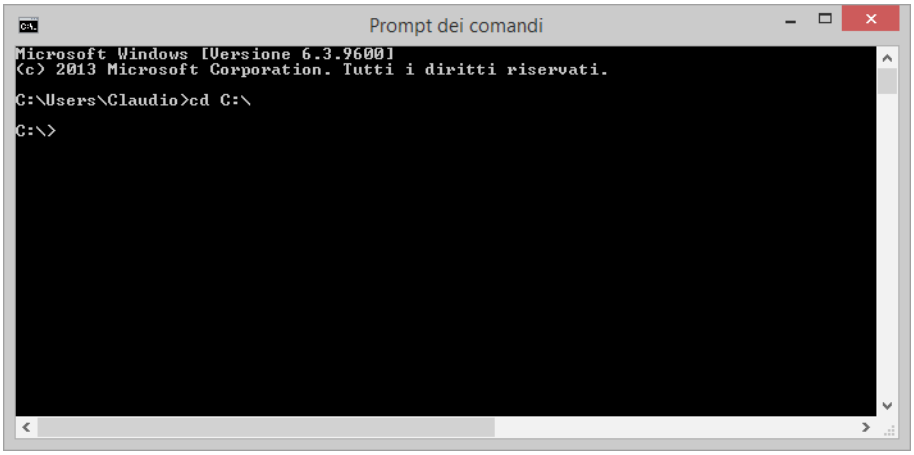


Figura C.1 - Prompt dei comandi (Windows 7).

Inizialmente, quando si apre la prompt dei comandi, dovreste trovarvi nella vostra "cartella utente" (vedi figura C.1). Nella tabella del prossimo paragrafo e nel paragrafo successivo è spiegato come spostarsi tra le tabelle.
Il sistema operativo DOS non aveva un'interfaccia grafica a finestre. Per navigare tra le cartelle, o per compiere un qualsiasi altro tipo di operazione sui file, bisognava digitare un comando. Segue una lista di comandi basilari che dovrebbero permettere al lettore di affrontare lo studio di questo libro senza problemi. I comandi eseguiti sulla prompt sono sempre **case insensitive** (non distinguono le lettere maiuscole e le lettere minuscole).

Ogni comando descritto deve essere seguito dalla pressione del tasto Invio (o Enter) sulla tastiera per avere effetto.

C.1 Tabella dei comandi DOS più comuni

Comando	Spiegazione
`cd ..`	Spostamento nella cartella che contiene la cartella di lavoro.

`cd nomeCartella`
 	
`cd percorsoCartelle`
	Serve per spostarsi in una cartella contenuta nella cartella di lavoro. Se il nome della cartella contiene spazi vuoti, bisogna includere il nome della cartella tra virgolette. Ecco qualche esempio:
	`cd ManualeDiJava`
	`cd "Manuale Di Java"`
	È possibile anche spostarsi in sotto cartelle:
	`cd cartella/sottoCartella/sottosottoCartella`
	Oppure spostarsi in percorsi che si trovano al di fuori della cartella dove siamo posizionati:
	`cd "../cartellaParallela/cartella il cui nome ha spazi"`

`dir`	Elenca il contenuto della cartella di lavoro con allineamento verticale.
`dir /w`	Elenca il contenuto della cartella di lavoro con allineamento orizzontale.
`dir /p`	Elenca il contenuto della cartella di lavoro con un allineamento verticale. Se i file da elencare eccedono la disponibilità visiva della finestra, vengono visualizzati inizialmente solo i file che rientrano nella finestra stessa. Alla pressione di un qualsiasi ulteriore tasto, il sistema visualizzerà la schermata successiva.

`cd nomeUnità:/...`	Lo spostamento in una cartella di un'altra unità non produrrà subito il risultato voluto. Supponiamo per esempio di trovarci nella cartella C:/Test, digitando il comando:
	`cd D:/Java8`
	non ci sposterà nella cartella di destinazione se non digitando successivamente il comando:
	`D:`
	Insomma per spostarci su un'altra unità c'è bisogno di un comando supplementare.

TAB	Il tasto tab permette di scrivere il nome di un file o cartella contenuto nella cartella dove si è posizionati. Premendo più volte il tasto tab, verranno proposti in ordine alfabetico tutti i nomi dei file e delle cartelle presenti nella cartella corrente. Per esempio se ci troviamo in una cartella dove ci sono i file HelloWorld.java e HelloWorld.class e la cartella Test. Premendo il tasto tab ripetutamente, verrà scritto prima HelloWorld.class, poi HelloWorld.java e poi Test. Poi il ciclo ricomincia.
	Possiamo anche iniziare a scrivere la parte iniziale del nome del file o cartella che vogliamo scrivere, e poi premere tab. In questo caso il sistema completerà la parola solo con le alternative compatibili. Per esempio se scriviamo:
	`cd t`
	e poi premiamo tab, il sistema completerà il nostro comando così:
	`cd Test`
	Oppure se scriviamo:
	`javac Hell`
	e poi premiamo tab, il sistema completerà il nostro comando così:
	`javac HelloWorld.class`
	premendo di nuovo tab avremo:
	`javac HelloWorld.java`
	Il sistema è abbastanza intelligente da capire che se premiamo tab dopo il comando:
	`cd`
	saranno proposti solo i nomi di cartelle (e non di file).

`control c` (ovvero la pressione contemporanea dei tasti ctrl e c)	Interrompe il processo in corso (utile nel caso di processo con ciclo di vita infinito).
Tasti Freccia Su e Freccia Giù	Permetteranno di navigare tramite gli ultimi comandi digitati.
`exit`	Viene chiusa la sessione della riga di comando.

C.2 Altre informazioni sulla prompt dei comandi

La pressione del tasto destro del mouse sulla barra del titolo, consentirà la **personalizzazione** della prompt dei comandi, facendo clic sul menu Proprietà. Dalla finestra che apparirà sarà possibile personalizzare colori, layout, caratteri e così via. Per **incollare** una stringa copiata esternamente sulla prompt di DOS, bisogna sempre premere il tasto destro sulla barra del titolo, poi fare clic sul sotto-menu Incolla del menu Modifica. Nel caso di Windows 10 è possibile anche incollare del testo utilizzando la combinazione di tasti (in inglese shortcut) ctrl e v.

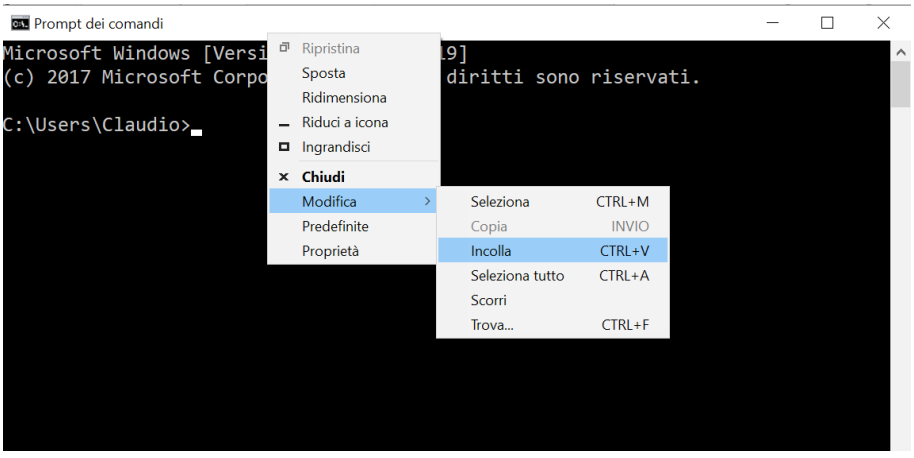


Figura C.2 - Menu della prompt dei comandi (Windows 10).

Se il vostro obiettivo è **copiare un percorso di una cartella** (che potrebbe anche essere molto lungo), potete anche digitare il comando:

```
cd
```

seguito da uno spazio. Poi trascinare dall'interfaccia a finestre di Windows, la cartella in cui volete spostarvi trascinandola all'interno della prompt dei comandi.
Per **copiare** una stringa dalla prompt dei comandi è invece necessario:

1. premere il tasto destro sulla barra del titolo;
2. fare clic sul sotto-menu Seleziona del menu Modifica (su Windows 10 è possibile utilizzare la combinazione di tasti ctrl e m);
3. selezionare la stringa da copiare trascinando il mouse con il tasto sinistro premuto sui caratteri da copiare;
4. fare clic sul sotto-menu Copia del menu Modifica o premere il tasto Invio (Enter).

A quel punto il testo selezionato sarà stato copiato, ed è pronto per essere incollato.
È possibile anche **fare delle ricerche** facendo clic sul sotto-menu Trova... del menu Modifica (su Windows 10 utilizzare la combinazione di tasti ctrl f).

Appendice D

Introduzione ai Design Pattern

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

✔ Saper definire cos'è un Design Pattern (unità D.1, D.2, D.3).

Applicare l'Object Orientation ai processi di sviluppo software complessi comporta non poche difficoltà. Uno dei momenti maggiormente critici nel ciclo di sviluppo è quello in cui si passa dalla fase di analisi a quella di progettazione. In tali situazioni bisogna compiere scelte di design particolarmente delicate, dal momento che potrebbero pregiudicare il funzionamento e la data di rilascio del software. In tale contesto (ma non solo) si inserisce il concetto di *Design Pattern*, importato nell'ingegneria del software direttamente dall'architettura. Infatti la prima definizione di pattern fu data da Cristopher Alexander, un importante architetto austriaco (insegnante all'università di Berkeley - California), che iniziò a formalizzare tale concetto sin dagli anni '60. Nel suo libro "Pattern Language: Towns, Buildings, Construction" (Oxford University Press, 1977) Alexander definisce un pattern come **una soluzione architetturale che può risolvere problemi in contesti eterogenei**.

La formalizzazione del concetto di Design Pattern (pattern di progettazione) è ampiamente attribuita alla cosiddetta *Gang of Four* (brevemente **GoF**). La "gang dei quattro" è costituita da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides che, nonostante provenissero da tre continenti diversi, in quattro anni di confronti catalogarono la prima serie di 23 pattern che costituirono il nucleo fondamentale della tecnica. Nel 1994 vide la luce il libro considerato la guida di riferimento per la comunità dei pattern: "Design Patterns: elements of reusable object oriented software" (Addison-Wesley). Altri autori in seguito hanno pubblicato testi che estendono il numero dei pattern noti.

D.1 Definizione di Design Pattern

I Design Pattern rappresentano soluzioni di progettazione generiche applicabili a problemi ricorrenti all'interno di contesti eterogenei. Consapevoli che l'asserzione precedente potrebbe non risultare chiara al lettore che non ha familiarità con determinate situazioni, cercheremo di descrivere il concetto presentando, oltre che la teoria, anche alcuni esempi di pattern all'interno del libro. In questo modo si potranno meglio apprezzare sia i concetti sia l'applicabilità.
L'idea di base, però, è piuttosto semplice. È risaputo che la bontà della progettazione è direttamente proporzionale all'esperienza del progettista. Un progettista esperto risolve i problemi che si presentano utilizzando soluzioni che in passato hanno già dato buoni risultati. La GoF non ha fatto altro che confrontare la propria (ampia) esperienza nel trovare soluzioni progettuali, scoprendo così alcune intersezioni evidenti. Siccome queste intersezioni sono anche soluzioni che risolvono spesso problemi in contesti eterogenei, possono essere dichiarate e formalizzate come Design Pattern. In questo modo si mettono a disposizione soluzioni a problemi comuni, anche ai progettisti che non hanno un'esperienza ampia come quella della GoF. Quindi stiamo parlando di una vera e propria miniera d'oro!

D.2 GoF Book: formalizzazione e classificazione

La GoF ha catalogato i pattern utilizzando un formalismo ben preciso. Ogni pattern, infatti, viene presentato tramite il nome, il problema a cui può essere applicato, la soluzione (non in un caso particolare) e le conseguenze. In particolare, ogni pattern viene descritto tramite l'elenco degli elementi che a loro parere sono maggiormente caratterizzanti:

1. *nome e classificazione*: importante per il vocabolario utilizzato nel progetto.
2. *Scopo*: breve descrizione di cosa fa il pattern e suo fondamento logico.
3. *Nomi alternativi* (se ve ne sono): molti pattern sono conosciuti con più nomi, perché magari "scoperti" da persone diverse che avevano dato loro nomi diversi.
4. *Motivazione*: descrizione di uno scenario che illustra un problema di progettazione e la soluzione offerta.
5. *Applicabilità*: quando può essere applicato il pattern.
6. *Struttura* (o *modello*): rappresentazione grafica delle classi del pattern mediante un linguaggio di notazione.

In questa libro utilizzeremo UML, ma sul libro dei GoF, la cui nascita è antecedente alla nascita di UML, sono utilizzati il linguaggio di notazione OMT (acronimo di "Object Modeling Technique" ovvero "tecnica di modellazione ad oggetti") creato da James Rumbaugh, e i diagrammi di interazione di Grady Booch.

7. *Partecipanti*: le classi/oggetti con le proprie responsabilità.
8. *Collaborazioni*: come collaborano i partecipanti per poter assumersi le responsabilità.
9. *Conseguenze*: pro e contro dell'applicazione del pattern.
10. *Implementazione*: come si può implementare il pattern.
11. *Codice d'esempio*: frammenti di codice che aiutano nella comprensione del pattern.

In questo libro si utilizzerà Java, ma sul libro la cui nascita è antecedente alla nascita di Java, vengono utilizzati C++ e SmallTalk.

12. *Pattern correlati*: relazioni con altri pattern.
13. *Utilizzi noti*: esempi di utilizzo reale del pattern in sistemi esistenti.

I 23 pattern presentati nel libro della GoF sono classificati in tre categorie principali, basandosi sullo scopo del pattern:

1. **pattern creazionali**: propongono soluzioni per creare oggetti.
2. **Pattern strutturali**: propongono soluzioni per la composizione strutturale di classi e oggetti.
3. **Pattern comportamentali**: propongono soluzioni per gestire il modo in cui vengono suddivise le responsabilità delle classi e degli oggetti.

Inoltre viene anche fatta una distinzione sulla base del raggio d'azione del pattern:

1. **class pattern**: offrono soluzioni tramite classi e sottoclassi in relazioni statiche tra loro.
2. **Object Pattern**: offrono soluzioni dinamiche basate sugli oggetti.

Possiamo riassumere i 23 pattern del libro della GOF tramite la seguente tabella.

		Scopo		
		Creazionale	**Strutturale**	**Comportamentale**
Raggio di azione	**Class**	**FactoryMethod**	**Adapter (class)**	**Interpreter** **Template Method**
	Object	**Abstract Factory** **Builder** **Prototype** **Singleton**	**Adapter (object)** **Bridge** **Composite** **Decorator** **Facade** **Flyweight** **Proxy**	**Chain of responsability** **Command** **Iterator** **Mediator** **Memento** **Observer** **State** **Strategy**

D.3 Esempi di pattern presenti nel libro

Esempi di pattern GOF sono descritti all'interno di *Manuale di Java 9*. In particolare, dopo avere introdotto la parola chiave `static` nel paragrafo 6.8.5, nel 6.8.6 viene descritto il pattern **Singleton**, uno dei più famosi ed utilizzati.
All'interno del paragrafo 9.6.3 viene invece presentata una soluzione basata sul **Factory Method**, all'interno di un modulo che parla di eccezioni. Anche in questo caso si tratta di un pattern famosissimo ma spesso utilizzato in maniera semplificata. Il pattern viene ripreso nel paragrafo 19.3.1, quando parliamo dei servizi creati con il `ServiceLoader`.
Il pattern **Thread Pool** viene invece descritto mediante una sua implementazione nel paragrafo 15.6.3.3 quando si parla di "interfacce Executors".
Stesso discorso per il pattern **Iterator**, che trova nell'interfaccia `Iterator` e le sue

implementazioni un'ottima rappresentazione. Il tutto viene descritto soprattutto nel modulo 17.2.2.2.
Lo spettacolare pattern strutturale **Decorator**, viene invece descritto in maniera abbastanza approfondita nel paragrafo 18.2, per descrivere la struttura della libreria base dell'Input-Ouput.
Il pattern architetturale **MVC** (acronimo di **Model View Controller**) viene accennato all'interno del paragrafo S.4.1.5 dedicato al linguaggio FXML di JavaFX. Viene anche spiegato brevemente nell'appendice Q dedicata a Swing/AWT, e più tecnicamente su un datato (ma ancora interessante) articolo che potete scaricare insieme al relativo codice all'indirizzo: http://www.claudiodesio.com/mvc.htm. Tale articolo contiene anche delle brevi descrizioni di vari pattern utilizzati all'interno dell'MVC, come lo **Strategy**, l'**Observer** e lo stesso **Factory Method**.
Infine proprio la già citata appendice Q dedicata a Swing/AWT e GUI in generale, oltre all'**MVC**, spiega anche il funzionamento del pattern comportamentale **Observer**, che è alla base del meccanismo di gestione degli eventi di Java, e mostra anche come le librerie AWT e Swing siano state costruite usando il pattern strutturale **Composite**.
Abbiamo solamente accennato al pattern noto come **DTO** (acronimo di **Data Transfer Object** ovvero "oggetto di trasferimento dati") nel paragrafo 5.3.1, riguardante un'introduzione all'architettura del software.

Appendice E

La variabile d'ambiente CLASSPATH

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

- ✔ Essere in grado di saper impostare correttamente la variabile d'ambiente CLASSPATH (unità E.1, E.3).
- ✔ Aver capito, saper creare ed utilizzare file JAR (unità E.2, E.3).

A volte capita di non riuscire ad eseguire un programma Java. Infatti potrebbe essere necessario utilizzare librerie esterne, oppure potrebbe capitare che anche trovandoci nella stessa directory dei nostri file eseguibili, questi non siano "visti" dalla virtual machine. In entrambi i casi è molto probabile che il problema derivi da una non corretta impostazione della variabile d'ambiente CLASSPATH.

E.1 CLASSPATH

La variabile d'ambiente CLASSPATH viene utilizzata al runtime dalla virtual machine per trovare le classi che il programma vuole utilizzare. Per capire meglio facciamo subito un esempio. Supponiamo di posizionare la nostra applicazione all'interno della directory C:\NostraApplicazione. Ora, supponiamo che la nostra applicazione voglia utilizzare alcune classi situate all'interno di un'altra directory, diciamo C:\AltreClassiDaUtilizzare. La virtual machine non esplorerà tutte le directory del disco fisso per trovare le classi che gli servono, ma cercherà solo nelle directory indicate dalla variabile d'ambiente CLASSPATH. In questo caso, quindi, è necessario impostare CLASSPATH sia sulla directory C:\AltreClassiDaUtilizzare, sia sulla directory C:\NostraApplicazione, per poter eseguire correttamente la nostra applicazione. Per raggiungere l'obiettivo ci sono due soluzioni:

1. impostare la variabile d'ambiente CLASSPATH direttamente da sistema operativo;
2. impostare CLASSPATH solo per la nostra applicazione.

La prima soluzione è altamente sconsigliata. Si tratta di impostare in maniera permanente una variabile d'ambiente del sistema operativo (il procedimento è identico a quello descritto nell'appendice C per la variabile PATH). Questo implicherebbe l'impostazione permanente di questa variabile, e impedirà l'esecuzione di un'applicazione da un'altra directory diversa da quelle referenziate dal CLASSPATH (a meno che non si imposti nuovamente la variabile CLASSPATH).
La seconda soluzione invece è più flessibile e si implementa specificando il valore della variabile CLASSPATH con una opzione del comando java. Infatti il flag -classpath (oppure -cp) del comando java permette di specificare proprio il classpath, ma solo per l'applicazione che stiamo eseguendo. Per esempio, se ci troviamo nella directory C:\NostraApplicazione e vogliamo eseguire la nostra applicazione, dobbiamo eseguire il seguente comando:

```
java -cp .;C:\AltreClassiDaUtilizzare miopackage.MiaClasseConMain
```

dove con l'opzione -cp abbiamo specificato che il classpath deve puntare alla directory corrente (specificata con ".") e alla directory C:\AltreClassiDaUtilizzare. È anche possibile utilizzare percorsi relativi. Il seguente comando è equivalente al precedente:

```
java -cp .;..\AltreClassiDaUtilizzare miopackage.MiaClasseConMain
```

Si noti che abbiamo usato come separatore il ";", dato che stiamo supponendo di trovarci su un sistema Windows. Su un sistema Unix-Like il separatore sarebbe stato ":".

E.2 File JAR

Non è raro referenziare dalla propria applicazione librerie di classi esterne. È possibile distribuire librerie di classi all'interno di directory, ma il formato più comune con cui vengono create librerie di classi è il formato **JAR**. Il termine JAR sta per "Java Archive" (in italiano "archivio Java"). Si tratta di un formato assolutamente equivalente al classico formato ZIP. L'unica differenza tra un formato ZIP e uno

JAR è che un file JAR deve contenere una directory chiamata META-INF, con all'interno un file di testo chiamato MANIFEST.MF. Questo file può essere utilizzato per aggiungere proprietà all'archivio JAR in cui è contenuto. Per creare un file JAR è quindi possibile utilizzare un utility come WinZIP o WinRar, aggiungendo il file MANIFEST.MF in una directory META-INF. Il JDK però offre un'alternativa più comoda: l'applicazione jar. Con il seguente comando:

```
jar cvf libreria.jar MiaDirectory
```

creeremo un file chiamato libreria.jar con all'interno la directory MiaDirectory e tutto il suo contenuto.

Sul sistema operativo Windows è possibile creare file jar eseguibili. Questo significa che, una volta creato il file jar con all'interno la nostra applicazione, sarà possibile avviarla con un tipico doppio clic del mouse, così come se fosse un file .exe. Il discorso sarà approfondito nell'appendice Q.

E.3 CLASSPATH e file JAR

Se volessimo utilizzare dalla nostra applicazione classi contenute all'interno di un file JAR, non direttamente disponibile nella stessa directory dove è posizionata l'applicazione, è sempre possibile utilizzare il CLASSPATH. In tal caso, per eseguire la nostra applicazione, dovremo utilizzare un comando simile al seguente:

```
java -cp .;C:\DirectoryConFileJAR\MioFile.jar miopackage.MiaClasseConMain
```

La JVM si occuperà di recuperare i file .class all'interno del file JAR in maniera automatica ed ottimizzata. Se volessimo utilizzare più file JAR, dovremo eseguire un comando simile al seguente:

```
java -cp .;C:\DirectoryConFileJAR\MioFile.jar;C:\DirectoryConJAR\
AltroMioFile.jar miopackage.MiaClasseConMain
```

Se si vuole utilizzare una libreria JAR senza impostare il CLASSPATH è possibile inserire il file JAR nella directory jre/lib/ext che si trova all'interno dell'installazione del . . .

. . . JDK. In questo caso però tutte le applicazioni che saranno eseguite tramite il JDK potranno accedere a tale libreria. Per quanto riguarda EJE, è integrato un gestore di CLASSPATH che può essere configurato dalle opzioni.

Nel caso i file JAR che ci interessano si trovino nella stessa directory, è anche possibile utilizzare le cosiddette "CLASSPATH wildcards"; ovvero, il simbolo "*" può essere utilizzato per referenziare tutti i file JAR all'interno di una certa directory. Questo significa che il seguente comando è equivalente al precedente:

```
java -cp .;C:\DirectoryConFileJAR\* miopackage.MiaClasseConMain
```

È possibile utilizzare CLASSPATH wildcards solo dalla versione 6 del linguaggio.

Appendice F

Approfondimento sugli import statici

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

- ✔ Capire quando conviene usare gli import statici e che tipo di problemi possono sorgere con il loro utilizzo (unità F.1, F.2).

Bisognerebbe utilizzare gli import statici solo in pochi casi. La vera utilità di questa caratteristica è infatti limitata solo a poche situazioni. In compenso l'interpretazione errata dell'utilizzo di questo meccanismo da parte di un programmatore, può facilmente rendere le cose più complicate!
Gli import statici permettono al programmatore di importare solo ciò che è statico all'interno di una classe. La sintassi è:

```
import static nomePackage.nomeClasse.nomeMembroStatico
```

È consigliabile utilizzare gli import statici quando si è interessati solo ai membri statici di una certa classe, ma non alle sue istanze. Il beneficio immediato che riceve il programmatore è poter utilizzare i membri statici importati staticamente, senza referenziarli con il nome della classe. Per esempio possiamo importate staticamente l'oggetto `System.out`:

```
import static java.lang.System.out;
```

In questo modo sarà possibile scrivere all'interno del nostro codice:

```
out.println("25774");
```

in luogo di:

```
System.out.println("25774");
```

Nel caso in cui nel nostro file non si dichiarino o importino staticamente altre variabili `out`, non c'è alcun problema tecnico che ci impedisca di utilizzare l'import statico. In questo modo potremo scrivere meno codice noioso e ripetitivo come `System.out` all'interno della nostra classe.

L'utilità di tale funzionalità è reale se e solo se il membro statico importato staticamente è utilizzato più volte all'interno del codice.

È possibile importare anche tutti i membri statici di una classe utilizzando una wildcard, ovvero il solito simbolo di asterisco. Per esempio, con il seguente codice:

```
import static java.lang.Math.*;
```

abbiamo importato tutti i membri statici della classe `Math`. Quindi all'interno del nostro codice potremo chiamare i vari metodi statici di `Math` senza referenziarli con il nome della classe che li contiene. Per esempio, supponendo che `x` ed `y` siano le coordinate di un punto bidimensionale, il calcolo della distanza del punto dall'origine, prima dell'avvento degli import statici, si poteva codificare nel seguente modo:

```
Math.sqrt(Math.pow(x,2) + Math.pow(y,2));
```

ma è possibile ottenere lo stesso risultato con minor sforzo usando un import statico:

```
sqrt(pow(x,2) + pow(y,2));
```

Possiamo anche importare staticamente solamente metodi statici. In tal caso vengono specificati solo i nomi di tali metodi e non l'eventuale lista di argomenti (verranno importati ovviamente tutte le versioni dell'eventuale overload del metodo). Per esempio:

```
import static java.sql.DriverManager.getConnection();
```

Si possono anche importare staticamente classi innestate statiche. Per esempio è possibile importare la classe innestata statica `LookAndFeelInfo` della classe `UIManager` (una classe che contiene informazione sul look and feel di un'applicazione Swing, cfr. appendice Q) con la seguente sintassi:

```
import static javax.swing.UIManager.LookAndFeelInfo;
```

In questo caso, trattandosi di una classe innestata statica e quindi referenziabile con il nome della classe che la contiene, sarebbe possibile importarla anche nella maniera tradizionale con la seguente sintassi:

```
import javax.swing.UIManager.LookAndFeelInfo;
```

L'import statico però evidenzia il fatto che la classe innestata sia statica e forse, in tali situazioni, il suo utilizzo è più appropriato.

F.1 Usare gli import statici

La domanda è: siamo sicuri che l'utilizzo degli import statici rappresenti sempre un vantaggio? La risposta è: dipende! Gli stili e i gusti nel mondo della programmazione sono a volte soggettivi. C'è chi trova utile evitare di referenziare le variabili statiche, perché in questo modo scrive meno codice e trova questa pratica sufficientemente espressiva. Non è il nostro caso, come più volte sottolineato in questo testo, preferiamo un nome lungo ed esplicativo ad uno breve ed ambiguo. Se non siamo abituati a leggere tutti gli import di una classe (che possono anche essere tanti), è naturale pensare che un metodo o una variabile non referenziata sia semplicemente dichiarata all'interno della classe. Potremmo perdere tempo e concentrazione in una situazione del genere. Tuttavia esistono alcune situazioni che giustificano pienamente l'utilizzo degli import statici.

F.1.1 Enumerazioni

Un caso, dove l'uso degli import statici risulta effettivamente utile, è l'utilizzo delle enumerazioni. Per esempio, consideriamo l'enum `TigerNewFeature` definita di seguito:

```
package com.claudiodesio.appf;

public enum TigerNewFeature {
    ANNOTATIONS, AUTOBOXING, ENUMERATIONS, FOREACH,
    FORMATTING, GENERICS, STATIC_IMPORTS, VARARGS
}
```

Inoltre consideriamo un metodo di un'ipotetica classe `ProgrammatoreJava`:

```
package com.claudiodesio.appf;
import java.util.*;

public class ProgrammatoreJava {

    private String nome;
```

```
    private List<TigerNewFeature> aggiornamenti;

    //...

    public ProgrammatoreJava(String nome, TigerNewFeature... features) {
        this.nome = nome;
        aggiornamenti = new ArrayList<>();
        aggiungiAggiornamenti(features);
    }

    public void aggiungiAggiornamenti(TigerNewFeature... features) {
        for (TigerNewFeature aggiornamento : features) {
            aggiornamenti.add(aggiornamento);
        }
    }
}
```

Per aggiungere più feature dell'enumerazione `TigerNewFeature` sfruttando il metodo `aggiungiAggiornamenti(TigerNewFeature... features)` dovremmo utilizzare il seguente codice:

```
ProgrammatoreJava pro = new ProgrammatoreJava("Claudio",
  TigerNewFeature.VARARGS, TigerNewFeature.FOREACH,
  TigerNewFeature.ENUMERATIONS, TigerNewFeature.GENERICS);
```

In questo caso l'utilizzo di un import statico è senz'altro appropriato:

```
import static com.claudiodesio.appf.TigerNewFeature.*;
//...
ProgrammatoreJava pro = new ProgrammatoreJava("Claudio", VARARGS,
  FOREACH, ENUMERATIONS, GENERICS);
```

Infatti abbiamo evitato inutili ripetizioni e snellito il codice. Contemporaneamente la leggibilità non sembra essere peggiorata.

F.1.2 Astrazione

Un'altra situazione dove è pienamente giustificato l'utilizzo degli import statici è prettamente legata al concetto di astrazione (cfr. capitolo 6). Spesso capita di avere a disposizione un'interfaccia che definisce diverse costanti statiche (spesso si tratta di codice già esistente). Consideriamo per esempio la seguente interfaccia:

```
package applicazione.db.utility;
public interface CostantiSQL {
    String GET_ALL_USERS = "SELECT * FROM USERS";
    String GET_USER = "SELECT * FROM USERS WHERE ID = ?";
    // Altre costanti...
}
```

Questa interfaccia può avere un senso in alcuni contesti. Infatti essa definisce costanti di tipo stringa contenenti tutti i comandi SQL che una certa applicazione definisce. Così si favorisce il riuso di tali comandi in varie classi. Ora supponiamo di dover creare una classe che utilizza ripetutamente le costanti di tale interfaccia: la soluzione solitamente più utilizzata in questi casi è implementare l'interfaccia:

```
package applicazione.db.logic;
import applicazione.db.utility.*;
import java.sql.*;
import java.util.*;

public class GestoreDB implements CostantiSQL{
    public Collection getUsers() throws SQLException{
        Collection users = null;
        Statement statement = null;
        //...
        //ResultSet rs = statement.executeQuery(GET_ALL_USERS);
        //...
        return users;
    }
    // Altri metodi...
}
```

Tuttavia la soluzione più corretta sarebbe utilizzare l'interfaccia, non implementarla. Infatti, se la implementassimo, sarebbe come se stessimo dicendo che `GestoreDB` "è un" `CostantiSQL` (cfr. paragrafo 7.5). Una soluzione corretta potrebbe essere la seguente:

```
package applicazione.db.logic;
import applicazione.db.utility.*;
import java.sql.*;
import java.util.*;

public class GestoreDB {
    public Collection getUsers() throws SQLException{
        Collection users = null;
        Statement statement = null;
        //...
        ResultSet rs = statement.executeQuery(CostantiSQL.GET_ALL_USERS);
        //...
        return users;
    }
    // Altri metodi...
}
```

Considerando che non abbiamo riportato tutti i metodi (potrebbero essere decine) l'ultima soluzione obbliga il programmatore a dover scrivere codice un

po' troppo ripetitivo. Benché inesatta inoltre, la prima soluzione ha un notevole vantaggio programmatico e un basso impatto di errore analitico. In fondo stiamo ereditando costanti statiche, non metodi concreti. Quindi la situazione è questa: seconda soluzione più corretta, prima soluzione più conveniente! Bene, in questo caso gli import statici mettono tutti d'accordo:

```
package applicazione.db.logic;
import static applicazione.db.utility.CostantiSQL.*;
import java.sql.*;
import java.util.*;

public class GestoreDB {
    public Collection getUsers() throws SQLException{
        Collection users = null;
        Statement statement = null;
        //...
        ResultSet rs = statement.executeQuery(GET_ALL_USERS);
        //...
        return users;
    }
    // Altri metodi...
}
```

La terza soluzione è corretta e conveniente.

F.2 Impatto su Java

L'introduzione degli import statici in Java 5, fu probabilmente dovuta all'introduzione delle enumerazioni. Sembra sia quasi una caratteristica fatta apposta per dire: "le enumerazioni sono fantastiche e, se utilizzate gli import statici, eviterete anche di scrivere il codice ripetitivo che ne caratterizza la sintassi". Una conseguenza negativa dell'utilizzo non ponderato degli import statici è la perdita dell'identità dei membri importati staticamente. L'eliminazione del reference, se da un lato può semplificare il codice da scrivere, dall'altro potrebbe dar luogo ad ambiguità. Questo può avvenire essenzialmente in due situazioni: importando staticamente membri con lo stesso nome da tipi diversi, o con il fenomeno dello **shadowing** delle variabili importate con le variabili locali, o dei metodi importati con i metodi d'istanza.

F.2.1 Ambiguità

Nel caso in cui importassimo staticamente metodi con lo stesso nome di altri metodi già presenti all'interno delle nostre classi, varrebbero le regole dell'overload (cfr. capitolo 8). Quindi, se importiamo staticamente metodi con lo stesso nome di quelli

definiti nelle nostre classi, dobbiamo essere sicuri di avere per essi firme (in particolare liste di parametri) differenti. In caso contrario il compilatore segnalerà l'errore "riferimento ambiguo al metodo" se essi fossero utilizzati all'interno del codice senza reference, così come consentono gli import statici. In tal caso, per risolvere il problema di compilazione bisogna obbligatoriamente referenziare i metodi. Lo stesso discorso vale anche per le variabili. Consideriamo il seguente esempio:

```
import static java.lang.Math.*;
import static javax.print.attribute.standard.MediaSizeName.*;

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

La compilazione del file precedente darà luogo al seguente output:

```
VariabiliAmbigue.java:8: error: reference to E is ambiguous
        System.out.println(E);
                           ^
  both variable E in Math and variable E in MediaSizeName match
1 error
```

Bisogna quindi obbligatoriamente referenziare la costante `E` perché presente come variabile statica sia nella classe `Math` che nella classe `MediaSizeName` per eliminare l'ambiguità e non ottenere un errore in compilazione, per esempio nel seguente modo:

```
System.out.println(Math.E);
```

Si noti che l'errore è segnalato solo perché è stata usata la costante `E`, se avessimo utilizzato altri membri statici delle classi `Math` e `MediaSizeName`, non avremmo ottenuto errori in compilazione. Anche nel caso avessimo importato esplicitamente solo le due variabili `E`, come segue:

```
import static java.lang.Math.E;
import static javax.print.attribute.standard.MediaSizeName.E;
```

il compilatore avrebbe segnalato l'errore solo nel caso di utilizzo della costante `E`.

Valendo le regole tradizionali dell'overload, nel caso importassimo staticamente i metodi `sort()` della classe `Arrays` (ne esistono decine di overload) e della classe `Collections`, . . .

. . . non avremmo nessun problema nel loro utilizzo. Infatti le firme dei metodi sono tutte diverse e l'overload risulterebbe "ampliato".

F.2.2 Shadowing

Un altro problema causato dal non referenziare le variabili importate staticamente è noto come **shadowing**. Il fenomeno dello shadowing si manifesta quando dichiariamo una variabile locale con lo stesso nome di una variabile che ha uno scope più ampio, come una variabile d'istanza. Come già visto nel capitolo 2, all'interno del metodo dove è dichiarata la variabile locale, il compilatore considera *più importante* la variabile locale. In tali casi, per utilizzare la variabile d'istanza, bisogna obbligatoriamente referenziarla (nel caso di variabili d'istanza con il reference `this`). Lo shadowing affligge non solo le variabili d'istanza ma anche quelle importate staticamente, come mostra il seguente esempio:

```
import static java.lang.System.out;
//...
public void stampa(PrintWriter out, String text) {
    out.println(text);
}
```

All'interno del metodo `stampa()` il reference `out` non è `System.out`, ma il parametro di tipo `PrintWriter`. Consideriamo adesso il seguente codice:

```
import static java.lang.Math.pow;

public class Shadowing {
    public static double pow(double d1, double d2) {
        return 0;
    }

    public static void main(String args[]) {
        System.out.println(pow(2,2));
    }
}
```

Per effetto dello shadowing il metodo locale `pow()` sarà considerato prioritario rispetto a quello importato staticamente. Possiamo giungere alla conclusione che importare staticamente un membro di una classe, risulta essere un'operazione che va gestita con attenzione.

Appendice G

Un esempio guidato alla programmazione ad oggetti

Obiettivi:

Al termine di questa appendice il lettore dovrebbe essere in grado di:

- ✔ Avere in mente cosa significa sviluppare un'applicazione in Java utilizzando i paradigmi della programmazione ad oggetti (unità G.1, G.2, G.3).
- ✔ Capire cosa significa testare il proprio software anche in maniera automatica con JUnit (unità G.4).
- ✔ Conoscere sommariamente la libreria "Java Logging" (unità G.4).

In questa appendice verrà simulata la creazione di un semplice programma, passo dopo passo. L'accento sarà posto sulle scelte e i ragionamenti che bisogna svolgere quando si programma ad oggetti. In questo modo forniremo un esempio di come affrontare, almeno per i primi tempi, i problemi della programmazione object oriented. Si tratta di un approccio didattico, e la soluzione finale non rappresenterà un esempio di buona programmazione. Piuttosto servirà per comprendere i ragionamenti che devono guidarci all'interno delle varie fasi dello sviluppo. L'esempio presentato in questa appendice è orientato al programmatore, e non ad altri ruoli (progettista, architetto, etc.) che pur hanno a che fare con l'Object Orientation. Per avere una visione più globale del ciclo di sviluppo del software vi rimandiamo al case study presentato alla fine del capitolo 5, e sviluppato progressivamente nei successivi capitoli. Nella seconda parte saranno affrontati argomenti importanti come il testing e introdurremo il logging. Come il capitolo 5, questa appendice è pensata per avvicinare il lettore alla sviluppo di software reale, questa volta concentrandosi più sulla programmazione che su altri aspetti del ciclo di sviluppo del software.

G.1 Perché questa appendice

Questa appendice è stata introdotta per dare al lettore una piccola ma importante esperienza, finalizzata al corretto utilizzo dei paradigmi della programmazione ad oggetti. Quando si approccia all'Object Orientation, l'obiettivo più difficile da raggiungere non è comprenderne le definizioni, che come abbiamo visto sono derivate dal mondo reale, ma farne il giusto uso all'interno di un'applicazione. Ciò che probabilmente potrebbe mancare al lettore è la capacità di scrivere un programma, che sicuramente non è cosa secondaria! Facciamo un esempio: se fosse richiesto di scrivere un'applicazione che simuli una rubrica, o un gioco di carte, od un'altra qualsiasi applicazione, lo sviluppatore si porrà ben presto alcune domande: "quali saranno le classi che faranno parte dell'applicazione?", "dove utilizzare l'ereditarietà?", "dove utilizzare il polimorfismo?" e così via. Sono domande cui è molto difficile rispondere, perché per ognuna di esse esistono tante risposte che sembrano tutte valide. Se dovessimo decidere quali classi comporranno l'applicazione che simuli una rubrica, potremmo decidere di codificare tre classi (`Rubrica`, `Persona`, e la classe che contiene il metodo `main()`), oppure cinque (`Indirizzo`, `Ricerca`, `Rubrica`, `Persona`, e la classe che contiene il metodo `main()`). Riflessioni approfondite sulle varie situazioni che si potrebbero presentare nell'utilizzare quest'applicazione (i cosiddetti *casi d'uso*) probabilmente suggerirebbero la codifica di altre classi. Una soluzione con molte classi sarebbe probabilmente più funzionale, ma richiederebbe troppo sforzo implementativo per un'applicazione che si può definire "semplice". D'altronde, un'implementazione che fa uso di un numero ridotto di classi costringerebbe lo sviluppatore ad inserire troppo codice in troppe poche classi. Ciò garantirebbe inoltre in misura minore l'utilizzo dei paradigmi della programmazione ad oggetti, giacché la nostra applicazione, più che simulare la realtà, cercherebbe di *arrangiarla*. La soluzione migliore sarà assolutamente personale, perché garantita dal buon senso e dall'esperienza.
È già stato accennato che un importante supporto alla risoluzione di tali problemi viene garantito dalla conoscenza di metodologie object oriented, o almeno dalla conoscenza di UML (cfr. appendici I e L e i vari capitoli del libro che introducono esempi per i vari diagrammi). In questa sede però, dove il nostro obiettivo principale è quello di apprendere un linguaggio di programmazione, non è consigliabile né fattibile introdurre anche altri argomenti tanto complessi. Sicuramente non può bastare a risolvere tutti i nostri problemi ma può rappresentare un utile supporto allo studio dell'Object Orientation. Sarà presentato invece un esercizio-esempio, che il lettore può provare a risolvere oppure direttamente a studiarne la soluzione. Con quest'esercizio ci poniamo lo scopo di operare attente osservazioni sulle scelte

fatte, per poi trarne conclusioni importanti. La soluzione sarà presentata con una **filosofia iterativa ed incrementale** (ad ogni iterazione nel processo di sviluppo sarà incrementato il software) così com'è stata creata, sul modello delle moderne metodologie orientate agli oggetti. In questo modo saranno esposti al lettore tutti i passaggi eseguiti per arrivare alla soluzione.

G.2 Caso di studio

Obiettivo (problem statement):
Realizzare un'applicazione che possa calcolare la distanza geometrica tra punti. I punti possono trovarsi su riferimenti a due o tre dimensioni.

Con quest'esercizio non realizzeremo un'applicazione utile; lo scopo è puramente didattico.

G.3 Risoluzione del caso di studio

Di seguito è presentata una delle tante possibili soluzioni. Non bisogna considerarla come una soluzione da imitare, bensì come elemento di studio e riflessione per applicazioni future.

G.3.1 Passo 1

Individuiamo le classi di cui sicuramente l'applicazione non può fare a meno (quelle che nel capitolo 5 abbiamo definito come **key abstraction** o **astrazioni chiave**). Sembra evidente che componenti essenziali debbano essere le classi che devono costituire il **dominio** di quest'applicazione. Codifichiamo le classi `Punto` e `Punto3D` sfruttando incapsulamento, ereditarietà, overload di costruttori e riutilizzo del codice:

```
public class Punto {
    private int x, y;

    //Costruttore senza parametri
    public Punto() {
    }

    //Overload di costruttori: costruttore con due parametri interi
    public Punto(int x, int y) {
        //riutilizzo del codice (il this è facoltativo)
        this.setXY(x, y);
    }
```

```
    public void setX(int x) {
        this.x = x; //Il this non è facoltativo
    }

    public void setY(int y) {
        this.y = y; //Il this non è facoltativo
    }

    public void setXY(int x, int y) {
        this.setX(x); //Il this è facoltativo
        this.setY(y);
    }

    public int getX() {
        return this.x; //il this è facoltativo
    }

    public int getY() {
        return this.y; //il this è facoltativo
    }
}

public class Punto3D extends Punto {
    private int z;

    //Costruttore senza parametri
    public Punto3D() {
    }

    public Punto3D(int x, int y, int z) {
        //Riuso di codice
        this.setXYZ(x, y, z);
    }

    public void setZ(int z) {
        this.z = z; //il this non è facoltativo
    }

    public void setXYZ(int x, int y, int z) {
        //Riuso del codice
        this.setXY(x, y);
        this.setZ(z); //il this è facoltativo
    }

    public int getZ() {
        return this.z; //il this è facoltativo
    }
}
```

Facciamo subito una prima osservazione: notiamo che pur di utilizzare l'ereditarietà *legalmente* abbiamo violato la regola dell'astrazione. Infatti abbiamo assegnato l'identificatore `Punto` ad una classe che si sarebbe dovuta chiamare `Punto2D`. Avrebbe senso chiamare la classe `Punto` solo dove il contesto dell'applicazione chiaro fosse più restrittivo. Per esempio in un'applicazione che permetta di fare disegni, la classe `Punto` così definita avrebbe avuto senso. Nel momento in cui il contesto è invece generale come nel nostro esempio, non è detto che un `Punto` debba avere due dimensioni. Ricordiamo che, per implementare il meccanismo dell'ereditarietà, lo sviluppatore deve testarne la validità mediante la cosiddetta relazione "is a" (cfr. paragrafo 6.5.1). Violando l'astrazione abbiamo potuto validare l'ereditarietà. Ci siamo infatti chiesti: "un punto a tre dimensioni è un punto?". La risposta affermativa ci ha consentito la specializzazione della classe `Punto`. Se avessimo rispettato la regola dell'astrazione non avremmo potuto implementare l'ereditarietà tra queste classi, dal momento che ci saremmo dovuti chiedere: "un punto a tre dimensioni è un punto a due dimensioni?". In questo caso la risposta sarebbe stata negativa. Questa scelta è stata compiuta non perché ci faciliti il prosieguo dell'applicazione, anzi, proprio per osservare come lo sviluppo diventa più difficoltoso se si parte violando le regole fondamentali. Nonostante tutto la semplicità del problema e la potenza del linguaggio ci consentiranno di portare a termine il compito. Contemporaneamente vedremo quindi come forzare il codice affinché soddisfi i requisiti. In generale comunque è bene cercare di rispettare tutte le regole che possiamo nelle fasi iniziali dello sviluppo, relegando eventuali *forzature* per risolvere malfunzionamenti della nostra applicazione alle fasi finali dello sviluppo.

La codifica di due classi come queste è stata ottenuta apportando diverse modifiche. Il lettore non immagini di ottenere soluzioni ottimali al primo tentativo! Questa osservazione varrà anche relativamente alle codifiche realizzate nei prossimi passi.

G.3.2 Passo 2

Individuiamo le funzionalità del sistema. È stato richiesto che la nostra applicazione debba in qualche modo calcolare la distanza tra due punti. Facciamo alcune riflessioni prima di buttarci sul codice. Distinguiamo due tipi di calcolo della distanza tra due punti: tra due punti bidimensionali e tra due punti tridimensionali. Escludiamo a priori la possibilità di calcolare la distanza tra due punti di cui uno sia bidimensio-

nale e l'altro tridimensionale. A questo punto sembra sensato introdurre una nuova classe (per esempio la classe `Righello`) con la responsabilità di eseguire questi due tipi di calcoli. Nonostante questa appaia la soluzione più giusta, optiamo per un'altra strategia implementativa: assegniamo alle stesse classi `Punto` e `Punto3D` la responsabilità di calcolare le distanze relative ai *propri* oggetti. Si noti che l'astrarre queste classi inserendo nelle loro definizioni metodi chiamati `dammiDistanza()` rappresenta un'altra palese violazione alla regola dell'astrazione stessa. Infatti in questo modo potremmo affermare che intendiamo un oggetto come un punto capace di *calcolarsi da solo* la distanza geometrica che lo separa da un altro punto. E tutto ciò non rappresenta affatto una situazione reale. Quest'ulteriore violazione dell'astrazione di queste classi permetterà di valutarne le conseguenze e, contemporaneamente, di verificare la potenza e la coerenza della programmazione ad oggetti.

Ricordiamo che la distanza geometrica tra due punti bidimensionali è data dalla radice della somma del quadrato della differenza tra la prima coordinata del primo punto e la prima coordinata del secondo punto, e del quadrato della differenza tra la seconda coordinata del primo punto e la seconda coordinata del secondo punto.

Di seguito è presentata la nuova codifica della classe `Punto`, che dovrebbe poi essere estesa dalla classe `Punto3D`:

```
public class Punto {
    private int x, y;
    ... //inutile riscrivere l'intera classe
    public double dammiDistanza(Punto p) {
        //quadrato della differenza delle x dei due punti
        int tmp1 = (x - p.x) * (x - p.x);
        //quadrato della differenza della y dei due punti
        int tmp2 = (y - p.y) * (y - p.y);
        //radice quadrata della somma dei due quadrati
        return Math.sqrt(tmp1 + tmp2);
    }
}
```

Anche non essendo tecnicamente obbligati, per il paradigma del riuso potremmo chiamare comunque i metodi `p.getX()` e `p.getY()` piuttosto che utilizzare direttamente le variabili d'istanza `p.x` e `p.y`. Nel capitolo 6 infatti, . . .

. . . abbiamo visto che anche i metodi getter potrebbero implementare un algoritmo prima di restituire la variabile (anche se lo standard di implementazione dei metodi getter non prevede algoritmi). In tal caso riutilizzeremo l'eventuale algoritmo, e se questo si evolvesse non dovremmo modificare la chiamata al metodo getter.

Notiamo come in un eventuale metodo `main()` sarebbe possibile scrivere il seguente frammento di codice:

```
//creazione di un punto di coordinate x = 5 e y = 6
Punto p1 = new Punto(5,6);
//creazione di un punto di coordinate x = 10 e y = 20
Punto p2 = new Punto(10,20);
//invocazione del metodo dammiDistanza sull'oggetto p1
double dist = p1.dammiDistanza(p2);
//stampa del risultato
System.out.println("La distanza è " + dist);
```

Abbiamo già ottenuto un risultato incoraggiante!

G.3.3 Passo 3

Si noti che il metodo `dammiDistanza()` ereditato nella sottoclasse `Punto3D`, ha bisogno di un override per avere un senso. Ma ecco che le precedenti violazioni della regola dell'astrazione iniziano a mostrarci le prime incoerenze. Infatti, il metodo `dammiDistanza()` nella classe `Punto3D`, dovendo calcolare la distanza tra due punti tridimensionali, dovrebbe prendere in input come parametro un oggetto di tipo `Punto3D`, nella maniera seguente:

```
public class Punto3D extends Punto {
    private int z;

    //... inutile riscrivere l'intera classe
    public double dammiDistanza(Punto3D p) {
        //quadrato della differenza della x dei due punti
        int tmp1 = (x - p.x) * (x - p.x);
        //quadrato della differenza della y dei due punti
        int tmp2 = (y - p.y) * (y - p.y);
        //quadrato della differenza della z dei due punti
        int tmp3 = (z - p.z) * (z - p.z);
        //radice quadrata della somma dei tre quadrati
        return Math.sqrt(tmp1 + tmp2 + tmp3);
    }
}
```

In questa situazione però, ci troveremmo di fronte ad un overload piuttosto che ad un override! Infatti, oltre al metodo `dammiDistanza(Punto3D p)`, sarà ereditato dalla classe `Punto` in questa classe anche il metodo `dammiDistanza(Punto p)`. Quest'ultimo tuttavia, come abbiamo notato in precedenza, non dovrebbe essere disponibile in questa classe, giacché rappresenterebbe la possibilità di calcolare la distanza tra un punto bidimensionale ed uno tridimensionale.
La soluzione migliore sembra allora *forzare* un override. Possiamo riscrivere il metodo `dammiDistanza(Punto p)`. Dobbiamo però considerare il reference `p` come parametro polimorfo (cfr. paragrafo 8.3.1) ed utilizzare il casting di oggetti (cfr. paragrafo 8.3.3) all'interno del blocco di codice, per garantire il corretto funzionamento del metodo:

```
public class Punto3D extends Punto {
    private int z;
    //... inutile riscrivere l'intera classe

    public double dammiDistanza(Punto p) {
        if (p instanceof Punto3D) {
            Punto3D p1=(Punto3D)p; //Casting
            //quadrato della differenza della x dei due punti
            int tmp1 = (getX()-p1.getX()) * (getX()-p1.getX());
            //quadrato della differenza della y dei due punti
            int tmp2 = (getY()-p1.getY()) * (getY()-p1.getY());
            //quadrato della differenza della z dei due punti
            int tmp3 = (z-p1.z) * (z-p1.z);
            //radice quadrata della somma dei tre quadrati
            return Math.sqrt(tmp1 + tmp2 + tpm3);
        }
        else {
            return -1; //distanza non valida!
        }
    }
}
```

Come avevamo anticipato, abbiamo utilizzato le caratteristiche avanzate del linguaggio per risolvere problemi che sono nati dal non progettare la nostra soluzione sfruttando correttamente la regola dell'astrazione. Il metodo `dammiDistanza(Punto p)` dovrebbe ora funzionare correttamente. Infine è assolutamente consigliabile apportare qualche modifica stilistica al nostro codice, al fine di garantire una migliore astrazione funzionale, e una maggiore leggibilità del codice:

```
public class Punto3D extends Punto {
    private int z;

    //... inutile riscrivere l'intera classe
    public double dammiDistanza(Punto p) {
```

```
        if (p instanceof Punto3D) {
            //Chiamata ad un metodo privato tramite casting
            return this.calcolaDistanza((Punto3D)p);
        }
        else {
            return -1; //distanza non valida!
        }
    }

    private double calcolaDistanza(Punto3D altroPunto) {
        //quadrato della differenza della x dei due punti
        int tmp1 =
          (this.getX() - altroPunto.getX()) *
          (this.getX() - altroPunto.getX());
        //quadrato della differenza della y dei due punti
        int tmp2 =
          (this.getY() - altroPunto.getY()) *
          (this.getY() - altroPunto.getY());
        //quadrato della differenza della z dei due punti
        int tmp3 =
          (this.getZ() - altroPunto.getZ()) *
          (this.getZ() - altroPunto.getZ());
        //radice quadrata della somma dei tre quadrati
        return Math.sqrt(tmp1 + tpm2 + tpm3);
    }
}
```

Java ha tra le sue caratteristiche anche una potente gestione delle eccezioni, che costituisce uno dei punti di forza del linguaggio (cfr. capitolo 9). Sicuramente sarebbe meglio sollevare un'eccezione personalizzata piuttosto che ritornare un numero negativo in caso di errore. Questa appendice dovrebbe essere letta subito dopo lo studio del capitolo 8, ma nel caso aveste già studiato il capitolo 9, potreste anche provare a realizzare una soluzione basata sulle eccezioni.

G.3.4 Passo 4

Adesso abbiamo la possibilità di iniziare a scrivere la classe contenente il `main()`. Il nome da assegnarle dovrebbe coincidere con il nome dell'applicazione stessa. Optiamo per l'identificatore `TestGeometrico` giacché, piuttosto che considerare questa un'applicazione completa, preferiamo pensare ad essa come un test per un nucleo di classi funzionanti (`Punto` e `Punto3D`) che potranno essere riusate in

un'applicazione reale.

```
public class TestGeometrico {
    public static void main(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. Questo package è stato progettato con una filosofia ben precisa, basata sul design pattern **Decorator** (per informazioni sui pattern cfr. appendice D; per informazioni sul pattern Decorator cfr. capitolo 18). Il risultato è un'iniziale difficoltà d'approccio all'argomento, compensata però da una semplicità ed efficacia una volta compresi i concetti su cui si basa. 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 tramite rete. Rimandiamo il lettore al capitolo 18 relativo all'input/output per i dettagli. Intanto utilizzeremo il meccanismo basato sugli argomenti del metodo `main()`, già presentato nel paragrafo 3.6.5. Codifichiamo finalmente la nostra classe del main, chiamandola quindi `TestGeometrico`. Sfruttiamo gli argomenti del `main()` ed un metodo della libreria standard (`Integer.parseInt()`) per trasformare una stringa in intero:

```
public class TestGeometrico {
    public static void main(String args[]) {
        //Conversione a tipo int di stringhe
        int p1X = Integer.parseInt(args[0]);
        int p1Y = Integer.parseInt(args[1]);
        int p2X = Integer.parseInt(args[2]);
        int p2Y = Integer.parseInt(args[3]);
        //Istanza dei due punti
        Punto p1 = new Punto(p1X, p1Y);
        Punto p2 = new Punto(p2X, p2Y);
```

```
        //Stampa della distanza
        System.out.println("i punti distano " + p1.dammiDistanza(p2));
    }
}
```

Possiamo ora eseguire l'applicazione (ovviamente dopo la compilazione) per esempio scrivendo a riga di comando:

```
java TestGeometrico 5 6 10 20
```

il cui output sarà:

```
i punti distano 14.866068747318506
```

Per eseguire quest'applicazione siamo obbligati a passare da riga di comando quattro parametri interi, per non ottenere un'eccezione. In ambito esecutivo, altrimenti, la Java Virtual Machine incontrerà variabili con valori indefiniti come `args[0]`.

G.3.5 Passo 5

Miglioriamo la nostra applicazione in modo tale che possa calcolare anche la distanza tra due punti tridimensionali. Introduciamo prima un test per verificare se è stato inserito il giusto numero di parametri in input: se i parametri sono quattro, viene calcolata la distanza tra due punti bidimensionali, se i parametri sono sei, si calcola la distanza tra due punti tridimensionali. In ogni altro caso viene presentato un messaggio esplicativo e termina l'esecuzione del programma, prevenendo eventuali eccezioni in fase di runtime.

```
public class TestGeometrico {
    public static void main(String args[]) {
        // dichiariamo le variabili locali
        Punto p1 = null, p2 = null;

        // testiamo se è stato inserito il giusto numero di parametri
        if (args.length == 4) {// caso punti bidimensionali
            //Conversione a tipo int di stringhe
            int p1X = Integer.parseInt(args[0]);
            int p1Y = Integer.parseInt(args[1]);
            int p2X = Integer.parseInt(args[2]);
            int p2Y = Integer.parseInt(args[3]);
            //Istanza dei due punti
            p1 = new Punto(p1X, p1Y);
            p2 = new Punto(p2X, p2Y);
        }
        else if (args.length == 6) {
            //Conversione a tipo int di stringhe
            int p1X = Integer.parseInt(args[0]);
```

```
            int p1Y = Integer.parseInt(args[1]);
            int p1Z = Integer.parseInt(args[2]);
            int p2X = Integer.parseInt(args[3]);
            int p2Y = Integer.parseInt(args[4]);
            int p2Z = Integer.parseInt(args[5]);
            //Istanza dei due punti
            p1 = new Punto3D(p1X, p1Y, p1Z);
            p2 = new Punto3D(p2X, p2Y, p2Z);
        }
        else {
            System.out.println("inserisci 4 o 6 parametri");
            System.exit(0); // Termina l'applicazione
        }
        //Stampa della distanza
        System.out.println("i punti distano p1.dammiDistanza(p2));
    }
}
```

Le classi `Punto` e `Punto3D` sembrano ora riutilizzabili "dietro" altre applicazioni... o forse no...

G.4 Introduzione al test e al logging

Ma siamo sicuri che il codice che abbiamo scritto sia corretto? Siamo proprio sicuri che la nostra semplice applicazione funzioni in qualsiasi caso? È possibile che si comporti in maniera non prevista in certe occasioni?
Beh, questo non lo potremo mai dire! In effetti l'applicazione perfetta non esiste. Forse in applicazioni semplici come la nostra è possibile riuscire a non introdurre malfunzionamenti, ma in un'applicazione reale un baco è sempre possibile e probabilmente inevitabile.

In generale si definisce "baco" (in inglese "bug") un comportamento non previsto di un software.

Come possiamo allora creare un'applicazione dignitosa che non presenti bug (o almeno che non presenti bug importanti)? Esistono diverse filosofie che permettono di ridurre drasticamente i malfunzionamenti del software. Esistono tecniche di programmazione e di gestione del ciclo di sviluppo del software che sono state studiate proprio per ridurre e gestire i bug. Esistono anche vere e proprie metodo-

logie che promettono codice a "zero bug", ma crediamo sia sufficiente per questo libro limitarci solo ad introdurre una tecnica relativamente semplice da imparare e sperimentare: lo Unit Testing (in italiano "test unitario"). Più avanti introdurremo brevemente anche il logging.

G.4.1 Unit Testing in teoria

Prima di iniziare a descrivere cos'è un test unitario, iniziamo a definire cosa è un test. Creare una classe con il main e fare delle prove in fondo è già fare un test, quindi chiariamoci prima le idee. Esistono varie tipologie di test, i più famosi sono:

- **Test Unitari** (in inglese "**Unit Test**")
- **Test di Integrazione** (in inglese "**Integration Test**")
- **Test di Sistema** (in inglese "**System Test**")

Ci concentreremo solo sul primo tipo di test: i test unitari. Per quanto riguarda le altre tipologie di test citate ci limiteremo semplicemente ad affermare che i test di integrazione dovrebbero assicurarsi che i vari componenti del software lavorino correttamente insieme quando connessi. Mentre per test di sistema solitamente si intendono i test che si dovrebbero fare utilizzando la stessa interfaccia che utilizzerà l'utente finale.

Esistono poi altre tipologie di test che però non sembra opportuno introdurre in questo libro.

Il test unitario è una tecnica che esiste da anni e che ha trovato il suo successo definitivo contemporaneamente all'ascesa di un software free ed open source denominato JUnit (per informazioni e download http://www.junit.org).

Grande fama allo Unit Test è stata anche propiziata dall'affermarsi di alcuni processi metodologici come "Extreme Programming" (XP), il cui principale autore (Kent Beck) è anche uno degli ideatori di JUnit. XP è oggi relativamente poco utilizzata, sopravanzata da altre metodologie come Scrum (per informazioni https://www.scrum.org/resources/what-is-scrum**). Ma storicamente XP è stato uno dei primi processi ad abbracciare il manifesto agile** (http://agilemanifesto.org/iso/it/manifesto.html)**.**

Questo software è integrato come plug-in nella maggior parte dei tool di sviluppo professionali come Eclipse e Netbeans (entrambi tool free ed open source).
L'idea che è alla base dello unit testing è davvero semplice. Se il nostro codice deve funzionare, testiamolo classe per classe progettando e scrivendo **casi di test**, ovvero del codice che testa il nostro codice! Spesso la prima idea che può venire in mente dopo aver letto la frase precedente è: "Come? Scrivere i test prima del programma stesso?" e la seconda è "Ma se devo scrivere del codice affinché il mio software non abbia bachi, chi mi assicura che il codice dei miei test non abbia a sua volta bachi?". Avete perfettamente ragione, ma questo non è tutto! Pensate che è convinzione e prassi di moltissimi sviluppatori, soprattutto coloro che praticano (o pensano di praticare) processi agili, scrivere sempre prima i test e poi la classe da implementare! Ma come è possibile? Il mondo ora gira all'incontrario? Ma a pensarci bene non è così strano... pensiamo un attimo alla storia dei linguaggi di programmazione. Agli albori della programmazione si programmava utilizzando delle schede perforate in un linguaggio chiamato Assembly. Questo linguaggio permette di accedere al sistema su cui si sviluppa in maniera illimitata, e benché sia un linguaggio potentissimo, è anche un linguaggio pericolosissimo. Non per niente è anche uno dei linguaggi preferiti dagli hacker per creare virus. Per farla breve, in seguito sono nati altri linguaggi con compilatori sempre meno permissivi, COBOL, C, C++ e infine Java (e in seguito altri). La robustezza di Java è anche garantita dal suo compilatore, che come abbiamo visto più volte nei capitoli già letti, è molto attento anche a situazioni che compilatori di altri linguaggi non controllano per niente. La teoria degli Unit Test infondo si ispira a questa semplice teoria: se vogliamo delle classi robuste, creiamo prima (ma volendo anche dopo) un piccolo compilatore per la classe stessa! Ora non sembra tutto più razionale? Bene, allora cerchiamo di capire in pratica come creare ed utilizzare i nostri test unitari. Forse così riusciremo anche a capire come non introdurre bachi nei test che scriviamo.
Se la nostra intenzione è quella di scrivere dei piccoli compilatori per le nostre classi, non dobbiamo fare altro che creare delle classi che verifichino il corretto funzionamento di quanto si è scritto (o magari quello che si vuole scrivere). Per ogni classe della nostra applicazione è buona norma crearne un'altra che la testi. Per ogni metodo della nostra classe bisogna creare un numero variabile di *casi di test* ovvero di metodi nella classe di test, ognuno dei quali chiama con input differenti il metodo da testare, e verifichi che l'output sia quello aspettato. Quindi il numero dei possibili bachi che il metodo da testare potrebbe creare nella nostra applicazione scende con l'aumentare dei casi di test verificati.

G.4.2 Unit Test in pratica con JUnit

Per esempio, prendiamo la nostra classe `Punto`. Potremmo chiamare la nostra classe di test `TestPunto`. Cerchiamo poi di testare l'unico metodo non banale di questa classe, il metodo `dammiDistanza()`. Abbiamo già asserito che per ogni metodo da testare dobbiamo individuare un numero variabile di casi test. Ovvero dovremo passare input diversi verificando che l'output del metodo sia quello atteso. Il fatto che i possibili input del metodo siano praticamente infiniti non significa però che dobbiamo creare infiniti casi di test! Per poter progettare tutti i test che ci consentiranno di considerare il metodo funzionante e robusto, possiamo utilizzare una semplice tecnica: **le classi d'equivalenza**. Volendo essere brevi possiamo dire che bisogna cercare di creare un caso di test per ogni classe di equivalenza di input del metodo. Per classe d'equivalenza intendiamo: dato l'insieme di tutti i possibili casi di test del metodo, è possibile raggruppare questi casi in sottogruppi (detti appunto classi d'equivalenza) i cui elementi sono accomunati da una relazione d'equivalenza, ovvero, che da un certo punto di vista tali elementi sono equivalenti. Questo significa per esempio, che se vogliamo verificare che la distanza del punto di coordinate x=1 e y=1 dal punto di coordinate x=1 e y=2 è 1, sarà inutile verificare che la distanza dal punto di coordinate x=1 e y=3 è 2. Questi due test sono legati da una relazione di equivalenza (quella di appartenere alla stessa ascissa) e pertanto fanno parte della stessa classe di equivalenza. Se scegliamo quindi di implementare solo il primo caso, allora dobbiamo scrivere il seguente codice:

```
import org.junit.Assert;
import org.junit.Test;

public class TestPunto {
    @Test
    public void testDammiDistanzaSullAscissa() {
        Punto p1 = new Punto(1,1);
        Punto p2 = new Punto(1,2);
        double distanza = p1.dammiDistanza(p2);
        Assert.assertTrue(distanza == 1);
    }
}
```

L'utilizzo di JUnit è particolarmente semplice. Come è possibile notare abbiamo essenzialmente utilizzato un'annotazione (`@Test`) e un metodo statico (`assertTrue()`) della classe `org.junit.Assert`. La nostra classe di test non ha un metodo `main()`. Semplicemente può contenere vari metodi i quali se annotati da `@Test`, saranno invocati automaticamente da JUnit quando sarà eseguito il comando:

```
java org.junit.runner.JUnitCore TestPunto
```

Attenzione che questo comando funzionerà solo se nella variabile CLASSPATH è incluso il file junit.jar. Per informazioni sull'impostazione del classpath fare riferimento all'appendice E. Se utilizziamo un IDE come Eclispe basterà eseguire JUnit dall'apposito menu.

Con l'ultima istruzione del metodo:

```
        Assert.assertTrue(distanza == 1);
```

JUnit genererà un messaggio di successo o un rapporto di errore, a seconda del fatto che la condizione `distanza == 1` sia verificata o meno.
Se avevate in mente una classe di test, probabilmente l'avevate concepita con due piccole differenze: un metodo `main()`, obbligatorio per eseguire una classe, e al posto dell'istruzione finale del metodo, un semplice `System.out.println(distanza)` per stampare il risultato. Nulla di male, se non per il fatto che c'è bisogno dell'occhio umano per verificare la correttezza del test. Con JUnit possiamo automatizzare tali controlli con le cosiddette asserzioni.

JUnit è nato quando Java non supportava ancora la parola chiave `assert` (cfr capitolo 9). Quindi quando parliamo di asserzioni in JUnit intendiamo una serie di metodi di test che hanno nomi come `assertTrue()` o `assertEquals()`, che hanno il compito di decidere se una certa condizione (o appunto asserzione) è verificata o meno.

In questo modo potremo creare in una classe di test, una serie di metodi di test. Questa classe di test potrà anche essere eseguita in futuro, magari dopo del tempo in cui le nostre classi sono state modificate. In questo modo dovremo solo verificare che i test vadano a buon fine senza dovere obbligatoriamente ritornare a leggere il codice scritto per capire cosa doveva fare. Per esempio, se tra qualche mese modificheremo il metodo `dammiDistanza()`, per testarne la correttezza ci basterà rieseguire la nostra classe di test. Nel caso di errore l'output d JUnit ci permetterà di andare ad investigare sul problema.

È anche possibile eseguire un insieme di classi di test con un solo comando.

Progettiamo ora altri casi di test per verificare la correttezza del metodo `dammiDistanza()`. Per individuare casi di test, talvolta è utile pensare a situazioni che se dipendesse da noi, cercheremo sempre di evitare. Per esempio, un caso di test interessante potrebbe essere quello che passa al metodo invece di un oggetto di tipo `Punto`, un valore `null`. Un altro caso potrebbe essere quello di calcolare la distanza tra due punti con le stesse coordinate. Potremmo anche verificare la robustezza della nostra soluzione passando al metodo `dammiDistanza()` un oggetto di tipo `Punto3D` (parametro polimorfo). Scriviamo questi test arricchendo la classe `TestPunto`:

```
import org.junit.Assert;
import org.junit.Test;

public class TestPunto {
    @Test
    public void testDammiDistanzaSullAscissa() {
        Punto p1 = new Punto(1,1);
        Punto p2 = new Punto(1,2);
        double distanza = p1.dammiDistanza(p2);
        Assert.assertTrue(distanza == 1);
    }
    @Test
    public void testDammiDistanzaConNull() {
        Punto p1 = new Punto(1,1);
        Punto p2 = null;
        double distanza = p1.dammiDistanza(p2);
        Assert.assertTrue(distanza == -1);
    }
    @Test
    public void testDammiDistanzaDalloStessoPunto() {
        Punto p1 = new Punto(1,1);
        Punto p2 = new Punto(1,1);
        double distanza = p1.dammiDistanza(p2);
        Assert.assertTrue(distanza == 0);
    }
    @Test
    public void testDammiDistanzaDaUnPunto3D() {
        Punto p1 = new Punto(1,1);
        Punto p2 = new Punto3D(1,2,2);
        double distanza = p1.dammiDistanza(p2);
        Assert.assertTrue(distanza == -1);
    }
}
```

Per ora limitiamoci a fare questi test. Per eseguire il test usufruiamo del tool Eclipse, un tool completo, professionale, free ed open source che ingloba anche JUnit con una relativa interfaccia. In figura G.1 possiamo osservare il risultato dell'esecuzione del nostro test.

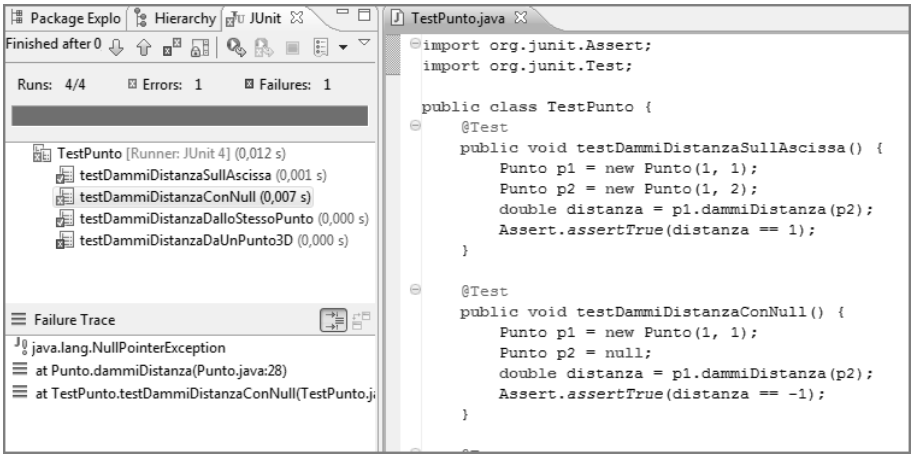


Figura G.1 - Output di JUnit su Eclipse.

Come è possibile notare i problemi ci sono e come! Il 50% dei nostri test sono falliti, ed abbiamo creato solo pochi test. In particolare il test denominato `testDammiDistanzaConNull()`, addirittura è terminato con un'eccezione non prevista (una `NullPointerException` cfr. capitolo 9).

Come esercizio il lettore dovrebbe provare a risolvere i problemi evidenziati da JUnit.

È possibile che il lettore che non ha esperienza di programmazione lavorative (o accademiche) non abbia idea di cosa significa sviluppare un software. Abbiamo solo poche righe a disposizione ma esistono libri giganteschi che descrivono questi argomenti. Quindi, per voi che non avete mai avuto esperienze pratiche di vero sviluppo, è importante sapere che quando inizia un progetto è molto probabile che non sarete gli unici sviluppatori. Ci saranno altre persone con cui condividere informazioni, tecniche e conoscenze e non tutte saranno sviluppatori, ma ognuno ha dei compiti e delle responsabilità (cfr. capitolo 5). La responsabilità dello sviluppatore non dovrebbe essere limitata al creare software a partire da requisiti. È importante anche cercare di assicurarsi che il codice scritto funzioni, sia documentato e che sia chiaro quali erano gli obiettivi che doveva implementare, per tutti gli sviluppatori che possono o potranno accedere in futuro al codice. Con i test unitari in qualche modo si rende anche esplicito quali sono gli obiettivi di una classe.
Non è detto che voi dobbiate per forza testare tutte le classi che creerete. Proba-

bilmente non ne sentite l'esigenza (anche se l'esempio di prima vi dovrebbe far riflettere) e probabilmente vorreste programmare in modo meno "estremo". Non siete obbligati a testare ogni classe e per gli standard di tempistica delle aziende italiane è spessissimo troppo dispendioso scrivere così tanti casi di test. Avere dei test automatizzati che permettono di testare la regressione del software (più codice introducete, più possibilità avete di introdurre bachi) è però un beneficio troppo grande per potervi rinunciare. Quindi ci sentiamo di consigliare la creazione di casi di test quantomeno di tutte le funzionalità più importanti se non volete testare classe per classe. In tal caso però non si può parlare proprio di "unit test", ma JUnit rimane comunque un valido strumento.

Ricordiamo che per scovare i bug, gli IDE mettono a disposizione lo strumento fondamentale denominato debugger, che abbiamo introdotto nel capitolo 5.

G.4.3 Java Logging

Come si risolve un bug? È probabile che per programmi semplici come quello dell'esempio di questa appendice, basti dare uno sguardo al codice o al massimo leggere gli eventuali output previsti dall'applicazione. Ma quando ci troveremo di fronte ad applicazioni reali non sarà così semplice risolvere i problemi. Prima di poter utilizzare il debugger, è necessario capire in quale parte del codice risiede il problema. Solo dopo aver capito più o meno dove si trova la causa del bug, è possibile impostare i punti di debug. Diventano così estremamente importanti i **messaggi di log**, ovvero i messaggi che facciamo stampare all'applicazione. Molti programmatori fanno grande uso del logging, ovvero la tecnica che ci permette di far stampare sulla console o in un file, delle informazioni riguardo le istruzioni che l'applicazione sta eseguendo. È stato deciso di non inserire all'interno del libro istruzioni sulla libreria nota come **Java Logging**, definita all'interno del package `java.util.logging`, incluso nel modulo denominato `java.logging`. Questo perché oltre a ragioni di spazio e tempo a disposizione, l'argomento è stato considerato utile soprattutto per ambienti Web-Enterprise. Tuttavia in questo paragrafo introdurremo solo brevemente l'argomento.

Gli **output di log** sono degli output che potremmo sia stampare sullo standard output (la console, esattamente come si fa con il `System.out.println()`) ma anche su un file o su di un flusso di rete. Il vantaggio essenziale rispetto ad una semplice stampa di output è che con un Java Logging è possibile specificare il *livello di severità*

del messaggio da stampare e filtrare quindi la stampa, specificando al runtime quale livello deve essere stampato. È possibile anche filtrare il livello specificando il nome del package o il nome delle singole classi. Per esempio, consideriamo il seguente codice:

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

public class LoggingExample {

    private final static Logger LOGGER =
      Logger.getLogger(LoggingExample.class.getName());

    public static void main(String args[]) {
        configureLog();
        LOGGER.info("Log a livello INFO");
        LOGGER.finest("Log a livello FINEST");
    }

    private static void configureLog() {
        Handler fileHandler = new FileHandler("file.log");
        Logger.getLogger("").addHandler(fileHandler);
        Logger.getLogger("LoggingExample").setLevel(Level.INFO);
    }
}
```

Abbiamo omesso volontariamente istruzioni per gestire le eccezioni per non confondere le idee al lettore.

La classe `LoggingExample` stampa in formato XML un log di livello `INFO`, all'interno di un file chiamato file.log. Invece l'istruzione di tipo `FINEST` non sarà stampata all'interno del file visto che `FINEST` è il livello di priorità più basso. L'output contenuto nel file file.log è:

```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2014-05-25T17:24:17</date>
  <millis>1401031457590</millis>
  <sequence>0</sequence>
  <logger>LoggingExample</logger>
  <level>INFO</level>
  <class>LoggingExample</class>
  <method>main</method>
  <thread>1</thread>
```

```
  <message>Log a livello INFO</message>
</record>
</log>
```

Per poter vedere stampati entrambi i log, dovremmo specificare nel metodo `configureLog()`, il livello `Level.FINEST` per la classe `LoggingExample`. Nella classe specificata abbiamo definito una costante statica `LOGGER` che abbiamo creato tramite il metodo statico `getLogger()` della classe `Logger`, passandogli il nome della classe `LoggingExample`. Si noti che in questo caso avremmo anche potuto scrivere:

```
Logger.getLogger("LoggingExample");
```

in luogo di:

```
Logger.getLogger(LoggingExample.class.getName());
```

ma se sbagliassimo a scrivere il nome della classe, il compilatore non ce lo segnalerebbe. Inoltre il nome della classe potrebbe anche cambiare nel tempo, ed una stringa non ci segnalerà nessun errore. Infine se avessimo dichiarato anche un package allora avremmo avuto più possibilità di sbagliare la stringa. Quindi è consigliata l'istruzione usata nell'esempio.
L'oggetto `LOGGER` ha dei metodi per ogni livello di log. Java Logging definisce sette livelli elencati nella tabella seguente in ordine di priorità, insieme al relativo metodo di utilizzo della classe `Logger`:

Livello della classe Level	Metodo di Logger
SEVERE	severe()
WARNING	warning()
INFO	info()
CONFIG	config()
FINE	fine()
FINER	finer()
FINEST	finest()

I metodi di logger hanno degli overload per poter passare come secondo parametro un'eccezione le cui informazioni saranno incluse nel record di log. È possibile anche usare equivalentemente il metodo `log()`, sfruttando il vantaggio che il livello può essere specificato come parametro. Per esempio:

```
LOGGER.severe("Eccezione a livello SEVERE", exception);
```

è equivalente a scrivere:

```
LOGGER.log(Level.SEVERE, "Eccezione a livello SEVERE", exception);
```

Nell'esempio possiamo anche notare come il metodo `configureLog()` definisca il livello di logging creando un oggetto `FileHandler`, e specificando il livello di log per la classe in questione.
Per chi fosse interessato ad approfondire l'argomento la documentazione è sufficientemente chiara, anche se in lingua inglese. Inoltre è possibile studiare il tutorial Oracle (aggiornato a Java 8, non ci sono novità da segnalare per Java 9) a questo indirizzo: http://docs.oracle.com/javase/8/docs/technotes/guides/logging/index.html.

Riepilogo

In questa appendice abbiamo visto un **esempio** di come si possa scrivere un'applicazione passo dopo passo. Lo scopo di questa appendice è stato quello di simulare lo sviluppo di una semplice applicazione e dei problemi che si possono presentare durante lo sviluppo. I problemi sono stati risolti uno a uno, a volte anche forzando le soluzioni. Si è fatto notare che il non rispettare le regole (per esempio l'astrazione) porta naturalmente alla nascita di problemi. Inoltre si è cercato di fare capire come si possano implementare, con esempi concreti, alcune caratteristiche object oriented.
Nei capitoli 6, 7 e 8 del manuale ci siamo dedicati esclusivamente al supporto che Java offre all'Object Orientation. Ma ovviamente non finisce qui! Ci sono tanti altri argomenti che vanno studiati, come le caratteristiche avanzate del linguaggio. È fondamentale capire che il lavoro di un programmatore che conosce Java ad un livello medio, ma in possesso di un buon metodo OO per lo sviluppo, vale molto più del lavoro di un programmatore con un livello di preparazione su Java formidabile, ma privo di un buon metodo OO. Il consiglio è quindi di approfondire le conoscenze sull'Object Orientation quanto prima, anche se, trattandosi di argomenti di livello sicuramente più astratto, bisogna sentirsi pronti per poterli studiare con profitto. Come sempre, il modo migliore per poter imparare certe argomentazioni è la pratica. Avere accanto una persona esperta (mentore) mentre si sviluppa è, dal nostro punto di vista, il modo migliore per crescere velocemente. La teoria da imparare è infatti sterminata, ma la cosa più complicata è mettere in pratica correttamente le tecniche descritte. Un'ulteriore difficoltà da superare è il punto di vista degli autori delle varie metodologie. Non è raro trovare in due testi, di autori diversi, consigli opposti per risolvere la stessa tipologia di problematica. Come già asserito, quindi,

il lettore deve sviluppare uno spirito critico nei confronti dei suoi studi; un tale approccio allo studio dell'OO richiede quindi una discreta esperienza di sviluppo.
Nella parte finale del capitolo abbiamo anche accennato alle pratiche di **testing** (sfruttando lo strumento **JUnit**). Abbiamo visto che creare prima i test e poi le classi che tali test devono testare non è un'idea così strana. Basta considerare le classi di test come dei mini compilatori per le nostre classi. JUnit utilizza il concetto di **asserzione**, che consente di scrivere codice che asserisce quale condizione deve essere verificata affinché un certo test sia superato. Il grande vantaggio di scrivere una suite di test per testare tutte le nostre classi (o anche solo alcune), consiste nel fatto che una volta scritta, è possibile eseguirla anche ogni giorno per controllare se il nostro sviluppo ha causato qualche regressione nel software (bug). Funzionerà effettivamente come compilatore delle classi.
Ancora di più se pensiamo ad una realtà aziendale, questo approccio risulterà molto utile.
Abbiamo infine solo introdotto il concetto di **logging**, parlando del framework **Java Logging**. Esso ci permette di rilasciare dei messaggi di log, che possono essere abilitati o disabilitati in base ad un livello che può essere impostato anche al runtime. In questo modo possiamo decidere di leggere solo i messaggi di log che ci interessano più facilmente.

Appendice H

Introduzione a XML

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

- ✔ Aver compreso cosa sono i linguaggi di markup (unità H.1).
- ✔ Saper definire brevemente XML, aver compreso come si struttura un documento XML, e aver appreso i concetti di documento XML ben formato e valido (unità H.2).

XML è l'acronimo di **eXtensible Markup Language**, che in italiano potremmo tradurre come **linguaggio di marcatura estensibile**. Solitamente però, anche noi italiani non traduciamo con "marcatura" il termine "markup", quindi parleremo di **linguaggio di markup**. Partiamo quindi prima da questa definizione.

H.1 Linguaggi di markup

Un **linguaggio di markup** è un linguaggio che permette di annotare documenti testuali in modo che essi siano leggibili sia dal punto di vista della macchina che dal punto di vista dell'utente umano. Solitamente le *marcature* sono costituite dai cosiddetti **tag** (in italiano **etichette**) che circondano parti del testo del documento proprio per etichettarle. I linguaggi di markup, più che altro si dovrebbero considerare metalinguaggi. Infatti i tag di un linguaggio di markup *annotano* il testo, associandogli informazioni che possono essere interpretate sia da un umano che da un software.

Esistono diversi linguaggi di markup, il primo fu inventato da IBM nel 1969, e si chiama **GML** (acronimo di **Generalized Markup Language**, ovvero **linguaggio di markup generalizzato**).

Il GML si evolse in seguito in **SGML** (acronimo di **Standard Generalized Markup Language**, ovvero **linguaggio di markup generalizzato standard**). Fu creato dall'ANSI (https://www.ansi.org) e fu standardizzato dall'ISO (https://www.iso.org/home.html) nel 1986. Si tratta di un metalinguaggio che consente di creare linguaggi di markup (come XML e HTML). I pregi di SGML erano la portabilità, la flessibilità, la potenza, la standardizzazione (ISO-8879) e il fatto di non essere una tecnologia proprietaria. I suoi difetti erano semplicemente due: troppo complesso per la maggior parte degli sviluppatori, e troppo pesante. Infatti, un documento SGML richiede necessariamente associati ad esso un Document Type Definition (DTD) ed un foglio di stile (Style-Sheet).
Tim Berners-Lee, fu ispirato da SGML quando creò l'**HTML**, acronimo di **HyperText Markup Language** (in italiano **linguaggio di markup per ipertesti**), linguaggio principale per scrivere ipertesti (pagine web).

Un "ipertesto" è una tipologia di documento (come una pagina web) che include, oltre che a semplice testo, collegamenti ad altre pagine.

L'HTML viene introdotto brevemente anche nell'appendice Q.
Come l'HTML, anche **XML** è un linguaggio di markup, e fu inizialmente definito nel 1996 da un gruppo di lavoro dell'SGML proprio per sostituire l'SGML, ma attualmente è supportato e definito dal consorzio W3C (https://www.w3.org).

H.2 Il linguaggio XML

XML invece è stato progettato per risultare un linguaggio semplice, robusto, estendibile e generico, utilizzabile tramite Internet. Il formato XML è basato sul testo, ed è quindi leggibile e facile da documentare. La caratteristica di robustezza è data dal fatto che un documento XML attraversa due fasi di verifica tra cui il test della validazione attraverso un altro file (DTD o Schema, cfr. paragrafo H.2.3). È estendibile perché è possibile creare tag personalizzati, e nuovi file per validarli e quindi nuove tecnologie. L'XML è stato anche la base su cui sono nate tantissime tecnologie, come per esempio i Web Services SOAP, il formato dei documenti di Microsoft Office, i feed RSS e tanti altri.

H.2.1 Documenti XML

Un documento XML è molto semplice. Di seguito ne creiamo uno con un solo tag chiamato `greeting`.

```
<?xml version="1.0"?>
<greeting>
    Hello World!
</greeting>
```

La prima riga viene detta **prologo**, ed è facoltativa (vedi paragrafo H.2.2). Poi segue il tag `greeting` che circonda il testo **"**Hello World!". Il nome del tag è sempre rinchiuso tra parentesi acute, ed un tag è opzionalmente composto da un tag di apertura e un tag di chiusura. Quest'ultimo è riconoscibile dal simbolo / che precede il nome del tag. Esistono altri tag che sono composti da un unico elemento caratterizzato da uno / che questa volta segue il nome del tag, e che rappresenta un tag di apertura e chiusura contemporaneamente. A volte questo tipo di tag viene detto **tag vuoto**, visto che non contiene altri tag o testo. Per esempio consideriamo il seguente esempio che astrae con XML una email:

```
<?xml version="1.0"?>
<email>
    <from>claudio@claudiodesio.com</from>
    <recipients>daxixlx.sxpxrx@gmail.com</recipients>
    <subject>Nuovo progetto</subject>
    <urgent/>
    <body format='text'>Ok, concludiamo! Ciao!</body>
</email>
```

Il tag `urgent` è una tipologia di tag costituito da un unico elemento di apertura e di chiusura. Il resto del codice è abbastanza chiaro, all'interno del tag `email` sono dichiarati tag che rappresentano il mittente (tag `from`), i destinatari (tag `recipients`), l'oggetto della mail (tag `subject`), l'urgenza della mail (tag `urgent`) e il corpo della mail (tag `body`) corredato anche da un attributo `format`, che indica il formato (in questo caso testuale) del corpo dell'email.

XML è case sensitive, e a schema libero.

H.2.2 Struttura di un documento XML

La **struttura** di un documento XML è sempre costituita da un **prologo** (opzionale ma raccomandato), seguita dai tag veri e propri che costituiscono l'elemento chiamato **document** (in italiano **documento**). Per esempio:

```
<?xml version="1.0"?>    <!-- prologo -->
<greeting>               <!-- inizio document -->
    Hello world!
</greeting>              <!-- fine document -->
```

H.2.2.1 Struttura del prologo

Il **prologo** ha la seguente struttura:

- dichiarazione XML (opzionale, ma raccomandata)
- informazione sulla versione
- informazione di Standalone
- informazione di Encoding (codifica)
- dichiarazione del tipo di documento (DTD) (esplicita e/o reindirizzata) (opzionale)
- commenti (opzionale)

Un esempio di prologo XML potrebbe essere:

```
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
```

Dove:

- `<?xml ... ?>`: è la dichiarazione XML
- `version`: definisce l'informazione sulla versione
- `standalone`: definisce l'informazione di validità
- `encoding`: definisce l'informazione di codifica

I commenti sono sempre inseriti all'interno di questa struttura di simboli:

```
<!-- Questo è un commento -->
```

e possono essere messi in qualsiasi parte del codice XML tra un tag e un altro.

H.2.2.2 Struttura del documento

Il **documento** ha la seguente struttura:

- elemento **root** (**radice**)
- altri elementi (opzionali)
- **attributi** (opzionali)
- testo (opzionale)
- commenti (opzionali)

Riprendiamo l'esempio di documento XML già visto nel precedente paragrafo 2.1

che astrae un'email:

```
<email>
    <from>claudio@claudiodesio.com</from>
    <recipients>daxixlx.sxpxrx@gmail.com</recipients>
    <subject>Nuovo progetto</subject>
    <urgent/>
    <body format='text'>Ok, concludiamo! Ciao!</body>
</email>
```

In questo caso:

- `<email>` è l'elemento root
- `<from>`, `<subject>`, `<recipients>`, `<urgent>` e `<body>` sono altri elementi
- `format` è l'attributo del tag `body`
- `Nuovo progetto` è il testo del tag `subject`, e `Ok, concludiamo! Ciao!` è il testo del tag `body`

H.2.3 Caratteristiche di un documento XML

Un documento XML deve essere **well formed** (in italiano **ben formato**). Un documento XML ben formato può essere inoltre dichiarato **valid** (in italiano **valido**) se soddisfa i vincoli specificati all'interno di un file che ne definisce la sintassi.

H.2.3.1 Documento XML well formed (ben formato)

Un documento XML si dice ben formato, quando:

- esiste un elemento root (radice)
- definisce elementi con tag di apertura (per esempio `<data>`) e di chiusura (per esempio `</data>`)
- definisce opzionalmente *elementi con tag speciali vuoti* (per esempio `<data/>`)
- i tag si possono innestare ma senza sovrapposizioni:
  - esempio di innesto NON valido:

```
<esterno>testo tag esterno<interno>testo tag interno</esterno></interno>
```

  - innesto valido:

```
<esterno>testo tag esterno<interno>testo tag interno</interno></esterno>
```

- i valori degli attributi possono essere racchiusi tra apici singoli o doppi. Per esempio:

```
      <font size="12">font test</font>
```

- i nomi degli elementi e degli attributi sono case-sensitive. I seguenti tag sono tutti differenti:`<pippo>`, `<PIPPO>` e `<Pippo>`

Si può verificare se un file XML sia ben formato aprendolo con un browser. In figura H.1 si può osservare il risultato dell'apertura del file ben formato che astrae un'email, con il browser Microsoft Edge:

```
<email>
    <from>claudio@claudiodesio.com</from>
    <recipients>daxixlx.sxpxrx@gmail.com</recipients>
    <subject>Nuovo progetto</subject>
    <urgent/>
    <body format='text'>Ok, concludiamo! Ciao!</body>
</email>
```

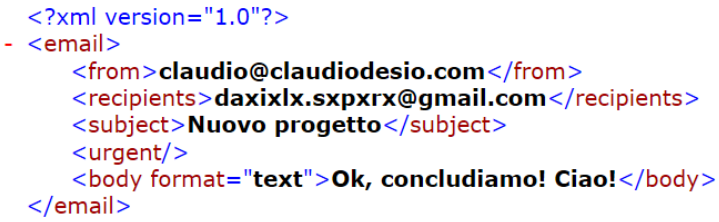


Figura H.1 - Visualizzazione di un documento ben formato all'interno del browser Microsoft Edge.

Si può notare come i tag possono essere anche compattati facendo clic sul simbolo – accanto al tag `email`. Se invece modifichiamo in maniera errata il file, per esempio eliminando il tag di chiusura `</subject>` :

```
<email>
    <from>claudio@claudiodesio.com</from>
    <recipients>daxixlx.sxpxrx@gmail.com</recipients>
    <subject>Nuovo progetto
    <urgent/>
    <body format='text'>Ok, concludiamo! Ciao!</body>
</email>
```

il browser non sarà più in grado di analizzare il codice XML e mostrerà la schermata rappresentata in figura H.2.

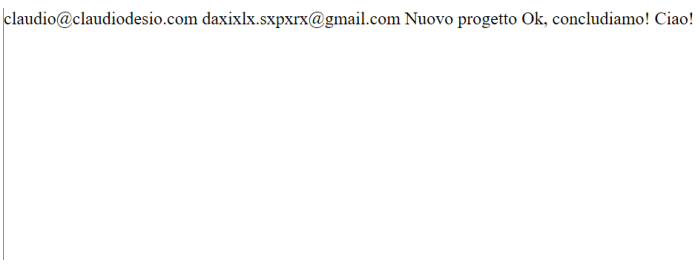


Figura H.2 - Visualizzazione di un documento NON ben formato all'interno del browser Microsoft Edge.

H.2.3.2 Documento XML valido

XML permette di creare tag a seconda delle necessità. In questo modo ci permette di astrarre qualsiasi tipo di concetto. A quel punto però, un programma che deve interagire con XML, deve conoscere la grammatica del documento per poterla interpretare. Per esempio supponiamo di creare un programma che trasforma documenti che astraggono email in vere email da inviare. È importante che tutti i documenti XML di tipo email abbiano le stesse caratteristiche e seguano delle regole, ovvero siano **validi** agli scopi del programma. Il meccanismo di validazione serve appunto per definire le regole che un documento XML deve rispettare. Tale meccanismo richiede:

1. la creazione di un file di validazione che definisca le regole di uno o più documenti XML (come vedremo un file DTD o XML Schema);
2. l'associazione del documento XML da validare al file di validazione;
3. l'esecuzione di un parser che analizza la validità del documento XML rispetto al suo file di validazione.

Esistono due tipi di file di validazione che si possono creare. Il primo tipo (caduto in disuso) è chiamato file **DTD** (acronimo di **Data Type Definition**, in italiano **definizione del tipo di dato**). Il secondo tipo si chiama invece **XML Schema**. Con un documento **DTD** possiamo definire la struttura di un documento XML. Consideriamo il seguente semplice file XML:

```
<?xml version="1.0"?>
<user>
    <name>Oscar</name>
    <surname>Wilde</surname>
    <id>8</id>
</user>
```

Il seguente file DTD, definisce la struttura del documento XML precedente:

```
<!DOCTYPE user
[
<!ELEMENT user (name,surname,id)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT id (#PCDATA)>
]>
```

Con `DOCTYPE` definiamo l'elemento radice (`email`) e con `ELEMENT` definiamo i vari tag. Con `(#PCDATA)` indichiamo al parser che si occuperà di validare il documento, che il tag contiene del testo, mentre con `EMPTY` che si tratta di un tag vuoto (`urgent`).
Se vogliamo utilizzare un XML Schema al posto di un file DTD, allora la sintassi cambia:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="user">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="surname" type="xs:string"/>
                <xs:element name="id" type="xs:integer"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Schema XML oggi rappresenta lo standard per la validazione dei documenti XML, visto che permette di specificare più vincoli rispetto ad un DTD, oltre ad avere una sintassi più intuitiva. Con Java è molto semplice validare file come vedremo nell'appendice N.

Appendice I

Introduzione allo Unified Modeling Language

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

✔ Saper definire UML e conoscere la sua storia (unità I.1, I.2, I.3, I.4).

Oggigiorno si sente parlare molto spesso di UML, ma non tutte le persone che parlano di UML sanno di che cosa realmente si tratti. Qualcuno pensa che sia un linguaggio di programmazione e quest'equivoco è dovuto alla parola "language". Qualcun altro pensa si tratti di una metodologia object-oriented e questo è probabilmente dovuto a cattiva interpretazione di letture non molto approfondite. Infatti si sente spesso parlare di UML congiuntamente a varie metodologie. Per definire quindi correttamente che cos'è UML, è preferibile prima definire per grandi linee che cos'è una metodologia.

I.1 Che cos'è UML (What)?

Una **metodologia** object-oriented, nella sua definizione più generale, potrebbe intendersi come una coppia costituita da un processo e da un linguaggio di modellazione.

Puntualizziamo che quella riportata è la definizione di "metodo". Una metodologia è tecnicamente definita come "la scienza che studia i metodi". Spesso però questi termini sono considerati sinonimi.

A sua volta un **processo** potrebbe essere definito come la serie di indicazioni riguardanti i passi da intraprendere per portare a termine con successo un progetto.
Un **linguaggio di modellazione** è invece lo strumento che le metodologie utilizzano per descrivere (possibilmente con dei grafici) tutte le caratteristiche statiche e dinamiche di un progetto.
UML non è altro che un linguaggio di modellazione. Esso è costituito per linee generali da una serie di diagrammi grafici i cui elementi sono semplici linee, ovali, rettangoli, omini stilizzati e così via. Questi diagrammi hanno il compito di descrivere in modo chiaro tutto ciò che durante un progetto potrebbe risultare difficile o troppo lungo con documentazione testuale.

I.2 Quando e dove è nato (When & Where)

A partire dai primi anni Ottanta la scena informatica mondiale iniziò ad essere invasa dai linguaggi orientati ad oggetti quali SmallTalk e soprattutto C++. Questo perché col crescere della complessità dei software, e della relativa filosofia di progettazione, la programmazione funzionale mostrò i suoi limiti e si rivelò insufficiente per soddisfare le sempre maggiori pretese tecnologiche. Ecco allora l'affermarsi della nuova mentalità ad oggetti e la nascita di nuove teorie, che si ponevano come scopo finale il fornire tecniche più o meno innovative per realizzare software sfruttando i paradigmi degli oggetti. Nacquero una dopo l'altra le metodologie object-oriented, in gran quantità e tutte più o meno valide. Inizialmente esse erano strettamente legate ad un ben determinato linguaggio di programmazione, ma la mentalità cambiò abbastanza presto. A partire dal 1988 iniziarono ad essere pubblicati i primi libri sull'analisi e la progettazione orientata agli oggetti. Nel '93 si era arrivati ad un punto in cui c'era gran confusione: analisti e progettisti esperti come James Rumbaugh, Jim Odell, Peter Coad, Ivar Jacobson, Grady Booch ed altri proponevano tutti una propria metodologia, ed ognuno di loro aveva una propria schiera di entusiasti seguaci. Quasi tutti gli autori più importanti erano americani ed inizialmente le idee che non provenivano dal nuovo continente erano accolte con sufficienza, e qualche volta addirittura derise. In particolare Jacobson, prima di rivoluzionare il mondo dell'ingegneria del software con il concetto di Use Case, fu denigrato da qualche autore americano, che definì infantile il suo modo di utilizzare omini stilizzati come fondamentali elementi dei suoi diagrammi. Probabilmente fu solo un tentativo di sbarazzarsi di un antagonista scomodo, o semplicemente le critiche provennero da fonti non certo lungimiranti.
Questi episodi danno però l'idea del clima di competizione di quel periodo storico, in cui si parlava di guerra delle metodologie. UML ha visto ufficialmente la luce a partire dal 1997 e i tempi dovevano ancora maturare...

I.3 Perché è nato UML (Why)

Il problema fondamentale era che diverse metodologie proponevano non solo diversi processi, il che può essere valutato positivamente, ma anche diverse notazioni. Era chiaro allora a tutti che non doveva e poteva esistere uno standard tra le metodologie. Infatti i vari processi esistenti erano proprietari di caratteristiche particolarmente adatte a risolvere alcune particolari problematiche. Quando si inizia un progetto, è giusto avere la possibilità di scegliere tra diversi stratagemmi risolutivi (processi). Il fatto che ogni processo fosse strettamente legato ad un determinato linguaggio di modellazione non rappresentava altro che un intralcio per i vari membri di un team di sviluppo. L'esigenza di un linguaggio standard per le metodologie era avvertita da tanti, ma nessuno degli autori aveva intenzione di fare il primo passo.

I.4 Chi ha fatto nascere UML (Who)

Ci pensò allora la **Rational Software Corporation** (oggi acquisita da IBM), che annoverava tra i suoi esperti **Grady Booch**, autore di una metodologia molto famosa all'epoca, nota come **Booch Metodology**. Nel '94 infatti, **James Rumbaugh**, creatore della **Object Modelling Tecnique** (**OMT**), probabilmente la più utilizzata tra le metodologie orientate agli oggetti, si unì al team di Booch alla Rational. Nell'Ottobre del '95 fu rilasciata la versione 0.8 del cosiddetto **Unified Method**. Ecco che allora lo svedese **Ivar Jacobson** nel giro di pochi giorni si unì anch'egli a Booch e Rumbaugh, iniziando una collaborazione che poi è divenuta storica. L'ultimo arrivato portava in eredità, oltre al fondamentale concetto di *Use Case*, la famosa metodologia **Object Oriented Software Engineering** (**OOSE**) conosciuta anche come **Objectory** (che in realtà era il nome della società di Jacobson oramai inglobata da Rational). Ecco allora che i **tres amigos** (così vennero soprannominati) iniziarono a lavorare per realizzare lo **Unified Software Development Process** (**USDP**) che poi venne rinominato semplicemente **Unified Process** (**UP**), e soprattutto al progetto UML. L'**Object Management Group** (**OMG**, http://www.omg.org), un consorzio senza scopi di lucro che si occupa delle standardizzazioni, le manutenzioni e le creazioni di specifiche che possano risultare utili al mondo delle tecnologie informatiche, nello stesso periodo inoltrò a tutti i più importanti autori una richiesta di proposta ("Request For Proposal", RFP) di un linguaggio di modellazione standard. Ecco che allora Rational, insieme con altri grossi partner quali IBM e Microsoft, propose la versione 1.0 di UML nell'ottobre del 1997. OMG rispose istituendo una "Revision Task Force" (RTF) capitanata da Cris Kobryn per apportare modifiche migliorative ad UML.
La versione attuale di UML è la 2.5.1, ma c'è molta confusione tra gli utenti di

UML. Infatti, la difficoltà di interpretare le specifiche, i punti di vista differenti da parte degli autori più importanti e l'ancoraggio di alcuni autori alle prime versioni del linguaggio (tres amigos in testa), rendono purtroppo il quadro poco chiaro. Effettivamente OMG ha come scopo finale far diventare UML uno standard ISO e per questo le specifiche sono destinate, più che agli utenti di UML, ai creatori dei tool di sviluppo di UML. Ecco perché le specifiche hanno la forma del *famigerato metamodello UML*. Ovvero, UML viene descritto tramite se stesso. In particolare, il metamodello UML è diviso in quattro sezioni e, giusto per avere un'idea, viene definito un linguaggio (l'**Object Constraint Language**, detto anche **OCL**) solo allo scopo di definire la sintassi senza ambiguità. Tutto questo nella apprezzabilissima ipotesi futura di creare applicazioni solo trascinando elementi UML uno sull'altro tramite tool di sviluppo, pratica che per chi scrive attualmente costituisce una parte consistente del ciclo di sviluppo del software. Consigliatissimo (anche nella bibliografia dell'appendice Z) per imparare UML, è il sempreverde best seller di Martin Fowler "UML Distilled". Libro conciso, pratico, "sincero" e pieno di preziosi riferimenti.

La sintassi dei diagrammi più importanti di UML sarà introdotta in modo schematico nella prossima appendice L.

Appendice L

UML Syntax Reference

Obiettivi:

Al termine di questa appendice il lettore dovrebbe essere in grado di:

✔ Saper consultare la seguente Syntax Reference per poter interpretare o creare semplici diagrammi UML (unità L.1).

In questa appendice vengono presentate delle tabelle schematiche della sintassi di UML 1.3. Anche se la versione è obsoleta, è possibile sfruttare ancora tutte le definizioni seguenti che costituiscono il nucleo fondamentale del linguaggio.

L.1 UML 1.3 Syntax Reference

Unified Modeling Language Syntax Reference		
Nome Diagramma	**Nomi degli Elementi**	
Use Case Diagram	Actor	Use Case
	Relationship Link	System Boundary
	Inclusion	Extension
	Generalization	Actor Generalization
Diagramma dei casi d'uso: rappresentano le interazioni tra il sistema e gli utenti del sistema stesso.		

Unified Modeling Language Syntax Reference

Class Diagram	Class / Object	Association / link	Navigability
	Attribute	Aggregation	Multiplicity
	Operation	Composition	Qualified Association
	Member Properties	Extension	Association Class
	Abstract Class / Interface	Implementation	Roles Names

Diagramma delle classi: descrive le classi che compongono il sistema e le relazioni statiche esistenti tra esse.
N.B.: quando un diagramma mostra gli oggetti del sistema, spesso ci si riferisce ad esso come diagramma degli oggetti (Object Diagram).

Component Diagram	Component	Dependency

Diagramma dei componenti: descrive i componenti software e le loro dipendenze.

Deployment Diagram	Node	Link

Diagramma di installazione (di dispiegamento, schieramento): mostra il sistema fisico.

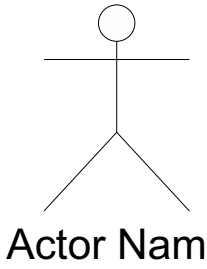
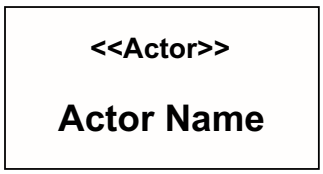
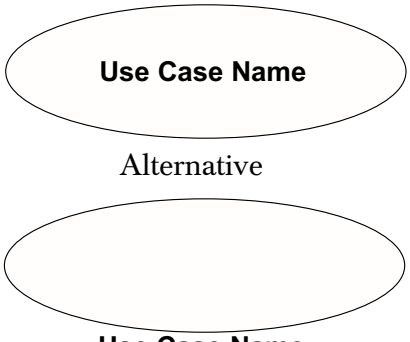
Interaction Diagrams: Sequence & Collaboration	Actor	Message
	Object	Asynchronous Message
	Creation	Life Line
	Destruction	Activity Line

Diagrammi di interazione: mostrano come gruppi di oggetti collaborano in un determinato lasso temporale.
Diagramma di sequenza: esalta la sequenza dei messaggi.
Diagramma di collaborazione: esalta la struttura architetturale degli oggetti.

Unified Modeling Language Syntax Reference		
State Transition Diagram	State	Transition
	Start	Action
	End	History
Diagramma degli stati (di stato): descrive il comportamento di un oggetto mostrando gli stati e gli eventi che causano i cambiamenti di stato (transizioni).		
Activity Diagram	Activity	Flow
	Branch/Merge	Fork/Join
	Swimlane	Other elements
Diagramma delle attività: descrive i processi del sistema tramite sequenze di attività sia condizionali sia parallele.		
General Purpose Elements & Extension Mechanism	Package	Iteration mark
	Stereotype	Condition
	Constraint	Tagged Value
Elementi generici e meccanismi d'estensione: elementi UML utilizzabili nella maggior parte dei diagrammi		

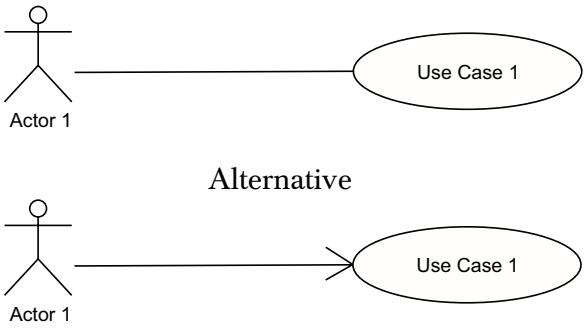
L.1.1 Use case diagram

Nella tabella seguente sono riportati gli elementi del diagramma dei casi d'uso (use case diagram).

Use Case Diagram Syntax Reference	
Nome Diagramma	**Nomi degli Elementi**
Actor	 Alternative 
Attore: ruolo interpretato dall'utente nei confronti del sistema. N.B.: un utente potrebbe non essere una persona fisica.	
Use Case	
Caso d'uso: insieme di scenari legati da un obiettivo comune per l'utente. Uno scenario è una sequenza di passi che descrivono l'interazione tra l'utenza ed il sistema. NB: è possibile descrivere scenari mediante diagrammi dinamici.	

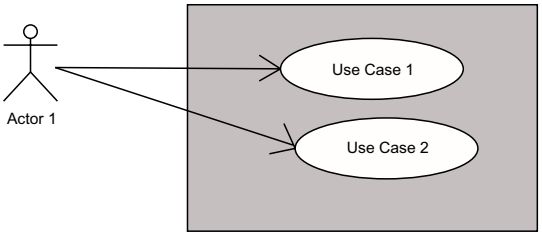
Use Case Diagram Syntax Reference

Relationship link



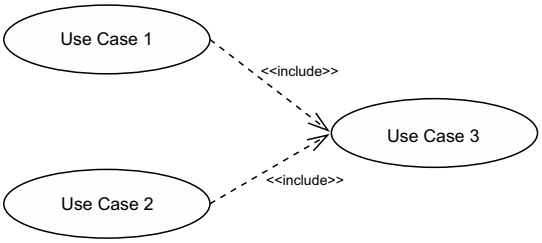
Associazione (o relazione): associa logicamente un attore ad un caso d'uso.

System Boundary



Sistema (o delimitatore del sistema): delimitatore del dominio del sistema.

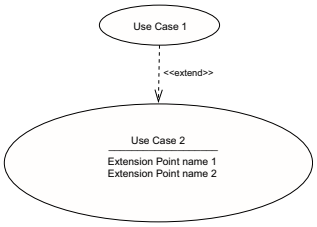
Inclusion



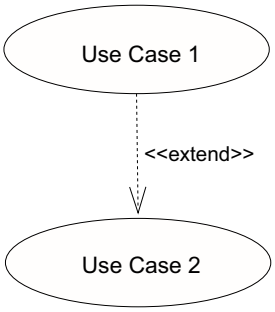
Inclusione: relazione logica tra casi d'uso, che estrae un comportamento comune a più casi d'uso.

Use Case Diagram Syntax Reference

Extension

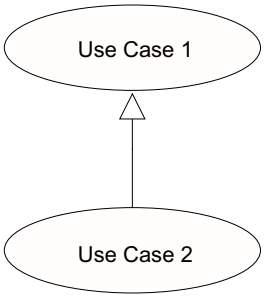


Alternative



Estensione: relazione logica che lega casi d'uso, che hanno lo stesso obiettivo semantico. Il caso d'uso specializzato raggiunge lo scopo aggiungendo determinati punti d'estensione, che sono esplicitati nel caso d'uso base.
Punto d'estensione: descrive un comportamento di un caso d'uso specializzato, non utilizzato dal caso d'uso base.

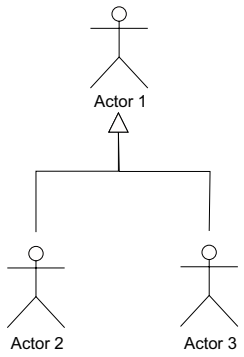
Generalization



Generalizzazione: relazione logica che lega casi d'uso, che hanno lo stesso obiettivo semantico. Il caso d'uso specializzato, raggiunge lo scopo aggiungendo nuovi comportamenti non utilizzati dal caso d'uso base, ma senza formalismi sintattici.

Use Case Diagram Syntax Reference

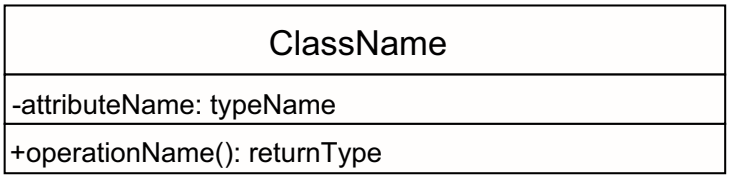
Actor generalization



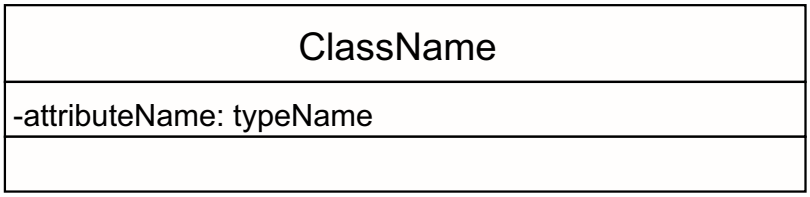
Generalizzazione tra attori: relazione logica che lega attori. Un attore che specializza un attore base, può relazionarsi a qualsiasi caso d'uso relazionato al caso d'uso base. Inoltre può relazionarsi anche ad altri casi d'uso, non relazionati con il caso d'uso base.

L.1.2 Class diagram

Nella tabella seguente sono riportati gli elementi del **diagramma delle classi** (**class diagram**).

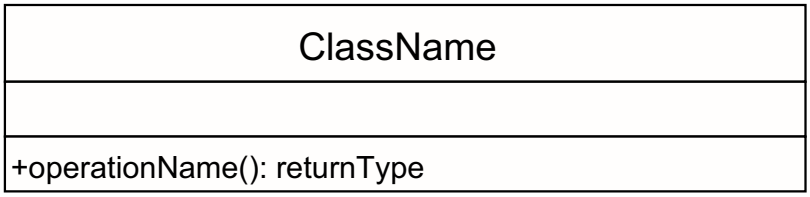
<table>
<tr><th colspan="2">Class Diagram Syntax Reference</th></tr>
<tr><td>Nome Diagramma</td><td>Nomi degli Elementi</td></tr>
<tr><td>Class & Object</td><td> 

Class
 

Object</td></tr>
<tr><td colspan="2">Classe: astrazione per un gruppo di oggetti che condividono stesse caratteristiche e funzionalità.
Oggetto: creazione fisica (o istanza) di una classe</td></tr>
<tr><td>Attribute</td><td> 
</td></tr>
<tr><td colspan="2">Attributo (o variabile d'istanza o variabile membro o caratteristica di una classe): caratteristica di una classe/oggetto.</td></tr>
</table>

Class Diagram Syntax Reference

Operation



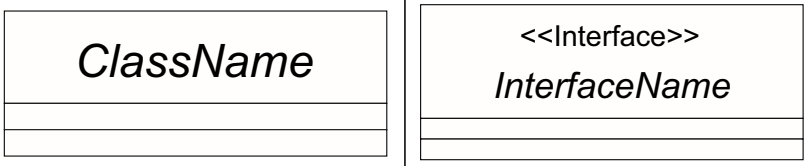
Operazione (o metodo o funzione membro): funzionalità di una classe/oggetto.

Member Properties

+ memberName “public member”
# memberName “protected member”
- memberName “private member”
memberName o $memberName “static member”
operation o operation {abstract} “abstract operation”
/attributeName “derived attribute”

Public, protected, private: modificatori di visibilità.
Membro statico (o membro della classe): membro della classe.
Operazione astratta: “signature” del metodo (metodo senza implementazione).
Attributo derivato: attributo ricavato da altri.

Abstract Class & Interface

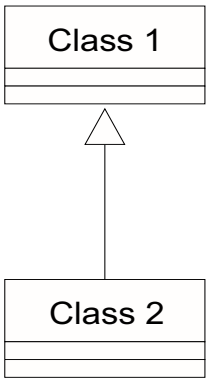


Abstract Class | Interface

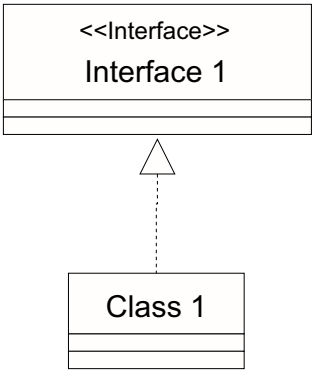
Classe astratta: classe che non si può istanziare e che può dichiarare metodi astratti.
Interfaccia: struttura dati non istanziabile che può dichiarare solo metodi astratti (e costanti statiche pubbliche).

Class Diagram Syntax Reference

Extension & Implementation



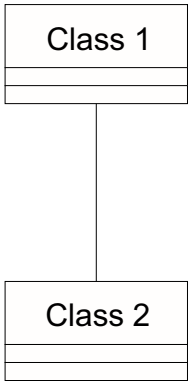
Extension



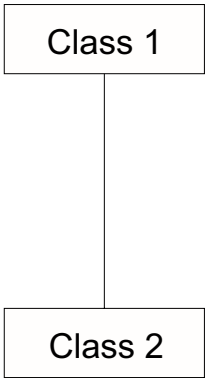
Implementation

Estensione: relazione di ereditarietà tra classi.
Implementazione: estensione per implementare metodi astratti di un'interfaccia.

Association



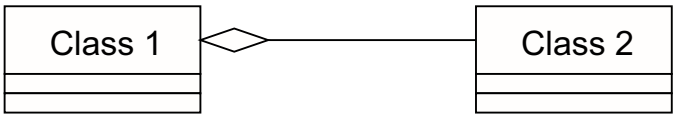
Association



Link

Associazione: relazione tra classi dove una utilizza i servizi (le operazioni) dell'altra.
Link: relazione tra associazione tra oggetti.

Aggregation



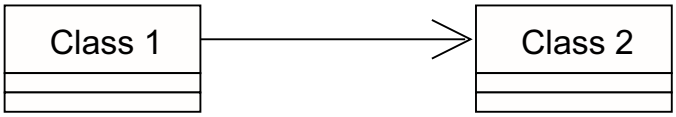
Aggregazione: associazione caratterizzata dal contenimento.

Class Diagram Syntax Reference

Composition

Composizione: aggregazione con cicli di vita coincidenti.

Navigability

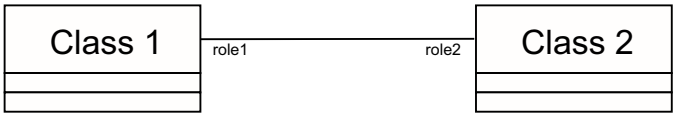


Navigabilità: direzione di un'associazione.

Multiplicity

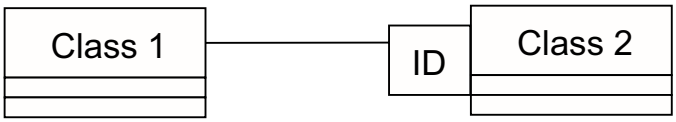
Molteplicità: corrispondenza tra la cardinalità del numero di oggetti delle classi coinvolte nell'associazione.

Role Names



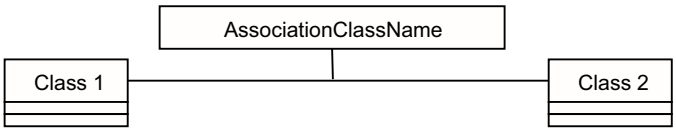
Ruoli: descrizione del comportamento di una classe relativamente ad un'associazione.

Qualified Association



Associazione qualificata: associazione nella quale un oggetto di una classe è identificato dalla classe associata mediante una "chiave primaria".

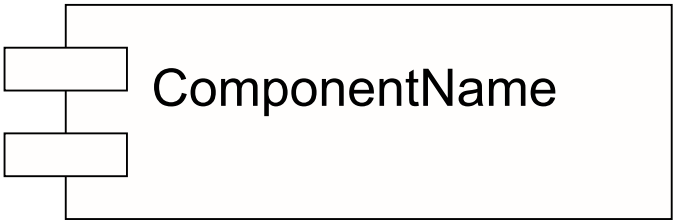
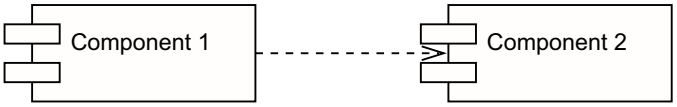
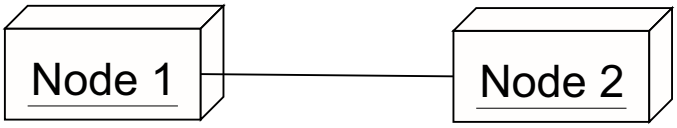
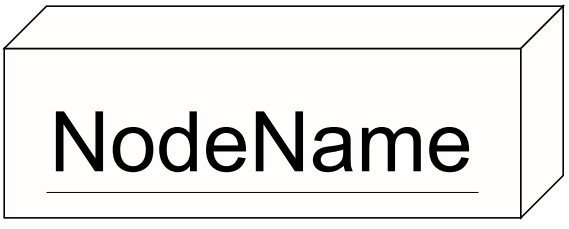
Association Class



Class d'associazione: codifica in classe di un'associazione.

L.1.3 Component e deployment diagram

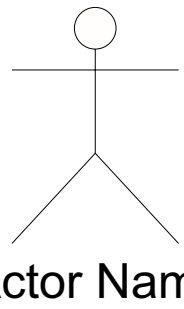
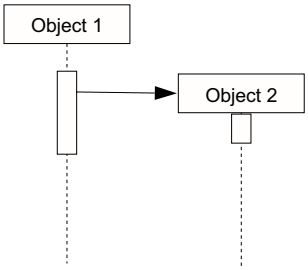
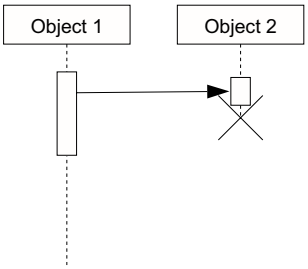
Nella tabella seguente sono riportati gli elementi del **diagramma dei componenti** (**component diagram**) e del **diagramma di installazione** (**deployment diagram**).

Component & Deployment Diagram Syntax Reference	
Nome Diagramma	**Nomi degli Elementi**
Component	
Componente: modulo software eseguibile, dotato di identità e con un'interfaccia ben specificata, di cui di solito è possibile uno sviluppo indipendente.	
Dependency	
Dipendenza: relazione tra due elementi di modellazione, nella quale un cambiamento sull'elemento indipendente avrà impatto sull'elemento dipendente.	
Link	
Associazione (o relazione): relazione logica nella quale un partecipante (componente o nodo o classe) utilizza i servizi dell'altro partecipante.	
Node	
Nodo: rappresentazione di una piattaforma hardware.	

L.1.4 Interaction diagram

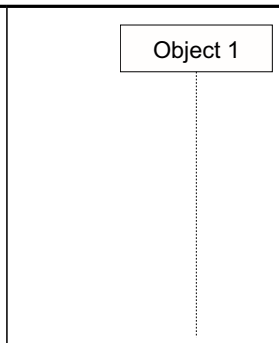
Nella seguente tabella sono riportati gli elementi dei **diagrammi di interazione** (**interaction diagrams**), ovvero il **diagramma di sequenza** (**sequence diagram**) e il **diagramma di collaborazione** (**collaboration diagram**).

Interaction Diagram Syntax Reference

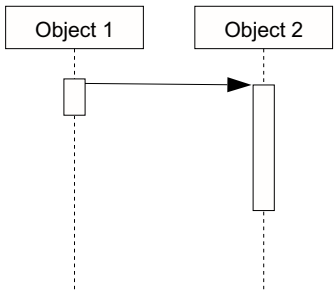
Nome Diagramma	Nomi degli Elementi
Actor	
Attore: ruolo interpretato dall'utente nei confronti del sistema. N.B.: un utente potrebbe non essere una persona fisica.	
Object	
Oggetto: creazione fisica (o istanza) di una classe	
Creation & Destruction	 Creation  Destruction
Creazione: punto d'inizio del ciclo di vita di un oggetto (può coincidere con una chiamata al costruttore dell'oggetto creato). Distruzione: punto terminale del ciclo di vita di un oggetto (può coincidere con una chiamata al distruttore dell'oggetto creato).	

Interaction Diagram Syntax Reference

Life Line & Activity Line



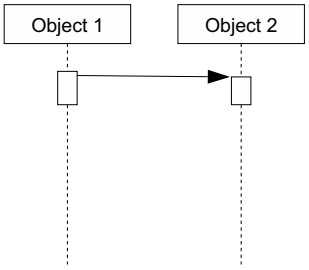
Life Line



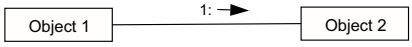
Activity Line

Linea di vita: linea di vita di un oggetto.
Linea di attività: linea di attività di un oggetto (rappresenta il blocco d'esecuzione di un metodo).

Message



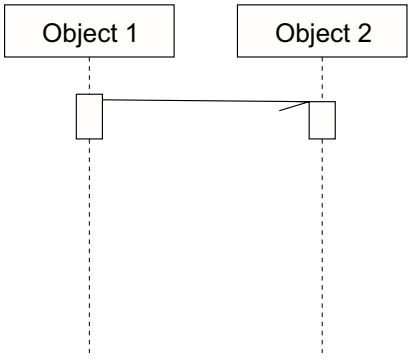
Sequence



Collaboration

Messaggio: messaggio di collaborazione tra oggetti (può coincidere con una chiamata al costruttore dell'oggetto di destinazione).

Asynchronous Message

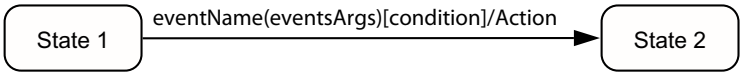
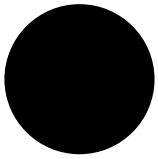
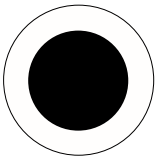
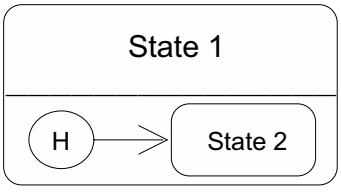


Asynchronuos Message

Messaggio asincrono: messaggio che può essere eseguito in maniera asincrona.

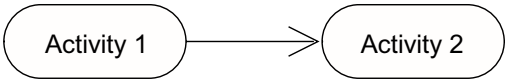
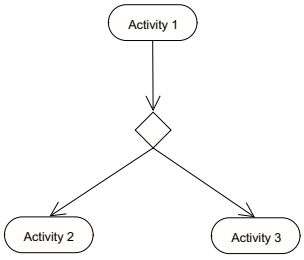
L.1.5 State diagram

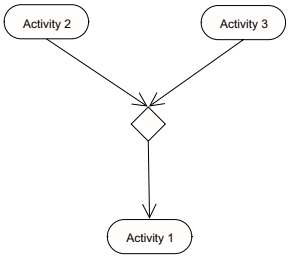
Nella seguente tabella sono riportati gli elementi del **diagramma degli stati** (**state diagram** o **state transition diagram**).

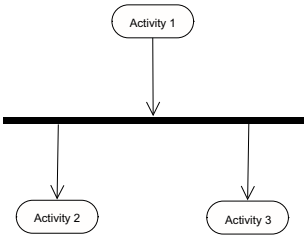
<table>
<tr><th colspan="3">State Transition Diagram Syntax Reference</th></tr>
<tr><th>Nome Diagramma</th><th colspan="2">Nomi degli Elementi</th></tr>
<tr><td>State</td><td colspan="2"> 
</td></tr>
<tr><td colspan="3">Stato: stato di un oggetto. Può essere caratterizzato da azioni (transizioni interne).</td></tr>
<tr><td>Transition</td><td colspan="2"> 
</td></tr>
<tr><td colspan="3">Transizione: attività che termina portando un oggetto in uno stato.</td></tr>
<tr><td>Start & End</td><td> 
Start</td><td> 
End</td></tr>
<tr><td colspan="3">Stato iniziale: punto iniziale di uno state transition diagram.
Stato finale: punto terminale di uno state transition diagram.</td></tr>
<tr><td>Action</td><td colspan="2"> 
</td></tr>
<tr><td colspan="3">Azione: attività che caratterizza uno stato.</td></tr>
<tr><td>History</td><td colspan="2"> 
</td></tr>
<tr><td colspan="3">Stato con memoria: stato capace di ripristinare situazioni precedenti.</td></tr>
</table>

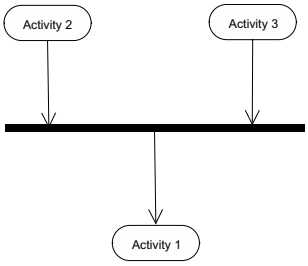
L.1.6 Activity diagram

Nella tabella seguente sono riportati gli elementi del **diagramma delle attività** (**activity diagram**).

<table>
<tr><th colspan="3">Activity Diagram Syntax Reference</th></tr>
<tr><th>Nome Diagramma</th><th colspan="2">Nomi degli Elementi</th></tr>
<tr><td>Activity</td><td colspan="2"> 
</td></tr>
<tr><td colspan="3">Attività: processo assolto dal sistema.</td></tr>
<tr><td>Flow</td><td colspan="2"> 
</td></tr>
<tr><td colspan="3">Flusso (di attività): insieme di scenari legati da un obiettivo comune per l'utente.</td></tr>
<tr><td>Branch & Merge</td><td> 

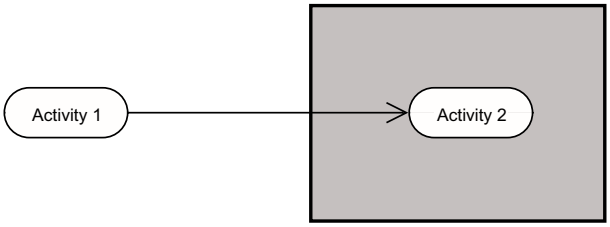
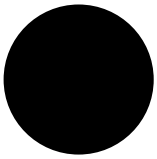
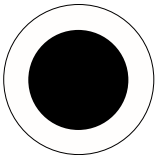
Branch</td><td> 

Merge</td></tr>
<tr><td colspan="3">Ramificazione: rappresenta una scelta condizionale.
Unione: rappresenta un punto di terminazione di blocchi condizionali.</td></tr>
<tr><td>Fork & Join</td><td> 

Fork</td><td> 

Join</td></tr>
<tr><td colspan="3">Biforcazione: rappresenta un punto di partenza per attività concorrenti.
Riunione: rappresenta un punto di terminazione per attività concorrenti .</td></tr>
</table>

Activity Diagram Syntax Reference

SwimLane		
Corsia: area di competenza di attività.		
Start & End	 Start	 End
Stato iniziale: punto iniziale di uno state transition diagram. Stato finale: punto terminale di uno state transition diagram.		
Object & State	 Object	 State
Oggetto: creazione fisica (o istanza) di una classe. Stato: stato di un oggetto. Può essere caratterizzato da azioni (transizioni interne).		

L.1.7 Elementi di uso generico

La tabella seguente riporta tutti gli elementi UML, che possono essere utilizzati su tutti i diagrammi.

General Purpose Elements Syntax Reference		
Nome Diagramma	**Nomi degli Elementi**	
Package & Stereotype	 Package	<<Stereotype>> Stereotype
Package: notazione che permette di ragruppare elementi UML. Stereotipo: meccanismo di estensione che permette di stereotipare costrutti non standard in UML.		
Constraint & Tagged value	{constraint} *Constraint*	*{key = value}* *Tagged Value*
Vincolo: meccanismo di estensione che permette di esprimere vincoli. Valore etichettato: vincolo di proprietà.		
Iteration Mark & condition	* *Iteration Mark*	*[condition]* *Condition*
Marcatore di iterazione: rappresenta un iterazione. Condition: rappresenta una condizione.		

Appendice M

Il modificatore `native`

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

✔ Capire come poter chiamare metodi nativi (unità M.1, M.2).

Nel libro abbiamo deciso di non includere questo argomento per problemi di priorità, i metodi nativi sono usati piuttosto raramente. Tuttavia per completezza, abbiamo deciso comunque di creare questa appendice per spiegare come funziona il modificatore `native`, che è l'unico che non è stato ancora trattato sino ad ora.

M.1 Java Native Interface

Il modificatore `native` serve per marcare metodi che possono invocare funzioni *native*. Questo significa che in Java è possibile scrivere applicazioni Java che invocano funzioni scritte in C/C++. Stiamo parlando di una delle tante tecnologie Java standard, nota come **JNI**, acronimo di **Java Native Interface**. Proponiamo come esempio la tipica applicazione che stampa la stringa `Hello World` tratta direttamente dal Java Tutorial. In questo caso però, un programma Java utilizzerà una libreria scritta in C++ per chiamare una funzione che stampa la scritta `Hello World`.

Se non si conosce il linguaggio C++, o almeno il C, si potrebbero avere difficoltà a comprendere il seguente esempio. Niente di grave, il nostro obiettivo rimane quello di imparare Java e non il C!

M.2 Esempio

Per prima cosa bisogna creare il file sorgente Java che dichiara un metodo nativo (il quale di default è anche astratto):

```
class HelloWorld {
    public native void displayHelloWorld();

    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

Notiamo come sia caricata una libreria in modo statico prima di tutto il resto (vedi blocco statico). La libreria denominata "`hello`", sarà caricata e ricercata con estensione .ddl su sistemi Windows, mentre verrà ricercato il file libhello.so su sistemi Unix. In particolare il caricamento della libreria dipende dalla piattaforma utilizzata e questo implica possibili problemi di portabilità.
Notiamo inoltre come il metodo `main()` vada a chiamare il metodo nativo `displayHelloWorld()` che, come un metodo astratto, non è definito.

M.2.1 Il tool javah

Dopo aver compilato il file bisogna utilizzare l'utility javah del JDK con la seguente sintassi:

```
javah -jni HelloWorld
```

In questo modo otterremo il seguente file header HelloWorld.h:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
    #endif
    /*
     * Class:     HelloWorld
     * Method:    displayHelloWorld
     * Signature: ()V
     */
```

```
    JNIEXPORT void JNICALL
      Java_HelloWorld_displayHelloWorld(JNIEnv *, jobject);

    #ifdef __cplusplus
}
#endif
#endif
```

L'unica cosa su cui porre l'attenzione è il nome della funzione C che viene denominata:

```
Java_HelloWorld_displayHelloWorld
```

ovvero, usa la sintassi:

```
Java_NomeClasse_nomeMetodo.
```

M.2.2 Codice nativo come libreria

Poi è necessario codificare il file C++ che viene chiamato HelloWorldImpl.c:

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>
JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj)
{
    printf("Hello World!\n");
    return;
}
```

Questo va compilato in modo tale da renderlo una libreria. Per esempio su Windows, se utilizziamo Microsoft Visual C++ 4.0, bisognerà eseguire il comando:

```
cl -Ic:\java\include -Ic:\java\include\win32 -LD HelloWorldImpl.c
  -Fehello.dll
```

per ottenere il file hello.dll.
Invece, su sistemi Solaris, è necessario specificare la seguente istruzione da una shell:

```
cc -G -I/usr/local/java/include - I/usr/local/java/include/solaris
  \ HelloWorldImpl.c -o libhello.so
```

per ottenere la libreria libhello.so.

M.2.3 Esecuzione

Non ci resta che mandare in esecuzione l'applicativo:

```
java HelloWorld
```

In questo modo la scritta:

```
HelloWorld!
```

Sarà stampata tramite il codice C, chiamato da codice Java.

JNI è una tecnologia che rende Java dipendente dal sistema operativo, con tutti gli svantaggi del caso, da utilizzare quindi solo se strettamente necessario.

Appendice N

Java e il mondo XML

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

- ✔ Essere in grado di utilizzare il supporto di Java al linguaggio XML, nelle situazioni più comuni (unità N.1).

XML (acronimo di **eXtensible Markup Language**) è un linguaggio oramai parte di qualsiasi tecnologia moderna. Trattasi del naturale complemento a Java per quanto riguarda il trasporto dei dati. Ovvero, come Java si distingue per l'indipendenza dalla piattaforma, XML può gestire il formato dei dati di un'applicazione qualsiasi sia il linguaggio di programmazione utilizzato. Gestire i dati in strutture XML significa infatti gestirli tramite semplici file testuali (o flussi di caratteri) che sono indipendenti dal linguaggio di programmazione o dalla piattaforma che li utilizza. Per informazioni su XML è possibile consultare migliaia di documenti su Internet oltre all'appendice H. La forza di questo linguaggio è essenzialmente dovuta alla sua semplicità. Java offre supporto a tutti i linguaggi o tecnologie basate su XML, da XSL a XPATH, da XSL-FO ai Web Services, dalla validazione con XML-schema a quella con il DTD, dall'esplorazione SAX a quella DOM e così via. In questa sede però, ci concentreremo essenzialmente sul supporto di Java all'utilizzo di XML come tecnologia di trasporto informazioni. Partiremo descrivendo i paradigmi di analisi delle librerie JAXP come DOM, SAX e StAX. Infine introdurremo anche il supporto alle trasformazioni XSLT.

XML è un linguaggio più giovane di Java, diventato in breve tempo "l'esperanto dell'informatica". Tramite esso, sistemi diversi riescono a dialogare, e questo è molto importante nell'era digitale. Oramai è difficile trovare un'applicazione professionale che non faccia uso in qualche modo di XML. In quest'appendice useremo un approccio basato su esempi per esplorare il supporto di Java all'integrazione con XML.

N.1 Modulo `java.xml`

Nel modulo `java.xml`, sono contenute tutte le librerie che supportano l'interazione tra Java e XML. I package che ci interesseranno di più sono `org.w3c.dom`, `org.xml.sax`, `javax.xml.stream`, `javax.xml.parsers`, `javax.xml.xpath` e `javax.xml.transform`, con i rispettivi sottopackage. I package `org.w3c.dom`, `org.xml.sax` e `javax.xml.stream` rappresentano essenzialmente le principali interfacce per il parsing (l'analisi) dei documenti XML. In particolare tali package contengono i tipi che realizzano le interfacce per XML note come **DOM**, **SAX** e **StAX**, e tutte queste, insieme alle **trasformazioni XSLT** definite soprattutto dal package `javax.xml.transform`, sono conosciute librerie **JAXP** (acronimo di **Java API for XML Processing**).
La libreria **DOM**, acronimo di **Document Object Model**, è basata sull'esplorazione del documento XML in maniera sequenziale partendo dal primo tag e scendendo nelle varie ramificazioni (si parla di **albero DOM**). È definita essenzialmente dalle interfacce del package `org.w3c.dom` e dei suoi sottopackage.
La libreria **SAX**, acronimo di **Simple API for XML**, supporta invece l'analisi di un documento XML basata su eventi. È definita per lo più dalle interfacce del package `org.xml.sax` e dei suoi sottopackage.
La libreria **StAX**, acronimo di **Streaming API for XML**, supporta un approccio che si pone a metà strada tra DOM e SAX, ed è simile agli stream di input-output. È definita soprattutto nei tipi del package `javax.xml.stream` e dei suoi sottopackage.
Il package `javax.xml.transform`, offre le classi e le interfacce che supportano le **trasformazioni XSLT**, che permettono di trasformare in altri formati il codice XML analizzato.
Il package `javax.xml.parsers` invece, oltre ad un'eccezione (`ParserConfigurationException`) e un errore (`FactoryConfigurationError`), definisce solo quattro classi (`DocumentBuilder`, `DocumentBuilderFactory`, `SAXParser` e `SAXParserFactory`) che rappresentano sostanzialmente delle factory per i principali concetti di JAXP. La situazione è simile a quella di JDBC (cfr. appendice R), dove dal `DriverManager` ricavavamo una `Connection`, dalla `Connection` uno `Statement` e così via. Con JAXP otterremo da un `DocumentBuilderFactory` un `DocumentBuilder`, da un `DocumentBuilder` un oggetto `Document` e dal `Document` vari altri oggetti fondamentali di XML come nodi e liste di nodi.
Il package `javax.xml.xpath` consentirà di rimpiazzare spesso righe di codice DOM grazie al linguaggio **XPATH**. Sicuramente si tratta del modo più veloce per recuperare i nodi di un documento XML.

Infine la libreria **JAXB** (acronimo di **Java Architecture for XML Binding**) si pone come alternativa a JAXP per l'analisi di testo XML, ed è definita soprattutto tramite i tipi definiti in `javax.xml.bind` e i suoi sottopackage. L'approccio di JAXB però è diverso, visto che consiste nel mappare il contenuto XML all'interno di un oggetto Java. Visto che gli argomenti sono davvero tanti, questa appendice si limiterà a fornire semplici esempi pratici per risolvere i problemi più comuni.

N.1.1 Document Object Model (DOM)

La libreria **DOM**, acronimo di **Document Object Model**, è basata sull'esplorazione del documento XML in maniera sequenziale partendo dal primo tag e scendendo nelle varie ramificazioni (si parla di **albero DOM**). Ha la caratteristica di permettere l'accesso a tag in maniera randomica all'interno del codice XML, nel senso che non è necessario obbligatoriamente esplorare l'intero file per poter accedere ad un certo tag ed al suo contenuto. Infatti verrà creata un'immagine in memoria del testo XML all'interno di un oggetto `Document`. È definita principalmente dalle interfacce del package `org.w3c.dom` e dei suoi sottopackage.

N.1.1.1 Creare un documento DOM a partire da un file XML

Un **documento DOM** è un oggetto di `org.w3c.dom.Document`, che sarà poi possibile analizzare tramite le funzionalità di DOM. Segue un esempio su come recuperare un documento DOM:

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
//...
    try {
        DocumentBuilderFactory factory =
          DocumentBuilderFactory.newInstance();
        factory.setValidating(false);
        Document doc =
          factory.newDocumentBuilder().parse(new File("nomeFile.xml"));
        //...
    } catch (Exception e) {
        //...
    }
```

Innanzitutto è necessario utilizzare la factory `DocumentBuilderFactory` per istanziare un `DocumentBuilder`. Il metodo `setValidating()` è stato utilizzato per dire all'oggetto factory di non validare il documento con l'eventuale DTD associato (cfr. appendice H). In realtà questa chiamata è superflua, perché `false`, rappresenta comunque il valore impostato di default. Infine il parse del file XML viene eseguito tramite il metodo `parse()`, a cui viene passato un'oggetto `File`. A

quel punto il nostro documento DOM, è rappresentato dall'oggetto `doc`.

> **Se volessimo creare un documento vuoto, basterebbe sostituire la chiamata al metodo `parse()` con la chiamata al metodo `newDocument()`.**

N.1.1.2 Recuperare la lista dei nodi da un documento DOM

Una volta ottenuto un oggetto di tipo `Document`, possiamo analizzarlo navigando nella sua alberatura. Per esempio:

```
NodeList list = doc.getElementsByTagName("*");
for (int i = 0; i < list.getLength(); i++) {
    Element element = (Element)list.item(i);
}
```

Sull'oggetto `doc` di tipo `Document` chiamiamo il metodo `getElementsByTagName()`, passandogli la wildcard `*`. In questo modo saranno restituiti (in un oggetto di tipo `NodeList`) solo i nodi principali, e non i loro sottonodi. Se vogliamo invece accedere a tutti i nodi, anche quelli innestati in profondità, non resta altro che creare un **metodo ricorsivo** (ovvero un metodo che richiama sé stesso) come il seguente:

```
public void findNode(Node node, int level) {
    NodeList list = node.getChildNodes();
    for (int i = 0; i < list.getLength(); i++) {
        Node childNode = list.item(i);
        findNode(childNode, level+1);
    }
}
```

Questo metodo va invocato con la seguente istruzione:

```
findNode(doc,0);
```

È possibile farlo perché `Document` implementa l'interfaccia `Node`. Una volta invocato il metodo sarà chiamato `getChildNodes()` sull'oggetto `node`. Gli oggetti `Node` ritornati sono immagazzinati in una `NodeList`. All'interno del `for`, si effettua un ciclo su tutti i nodi, e richiamiamo lo stesso metodo `findNode()` ricorsivamente, passandogli in input il nodo corrente e la variabile `level` avanzata di un'unità. Quest'ultima viene utilizzata solo come informazione per conoscere il livello di ramificazione del nodo che si sta analizzando. Per ogni nodo quindi si visitano i nodi innestati.

Purtroppo la libreria non è molto pratica da usare, risultando ai più piuttosto complessa.

Per le specifiche DOM ogni nodo è equivalente ad un altro. Questo significa che anche i commenti e il testo di un nodo (viene detto `TextNode`) sono a loro volta nodi. Lo stesso oggetto `Document` è considerato un nodo.

N.1.1.3 Recuperare particolari nodi

Per recuperare l'elemento radice di un documento XML ci sono due modi. Il primo è molto semplice e consiste nel chiamare il metodo `getDocumentElement()` sull'oggetto `Document`:

```
Element root = doc.getDocumentElement();
```

L'interfaccia `Element` implementa `Node`.

Il secondo metodo consiste nello scorrere l'albero DOM del documento (come nell'esempio del paragrafo precedente) e fermarsi al primo elemento di tipo `Element`. Questo controllo è obbligatorio altrimenti, a seconda del documento XML, si potrebbe recuperare un commento o la dichiarazione del `DocumentType`:

```
Element root = null;
NodeList list = doc.getChildNodes();
for (int i = 0; i < list.getLength(); i++) {
    if (list.item(i) instanceof Element) {
        root = (Element)list.item(i);
        break;
    }
}
```

Se la necessità è recuperare solo l'elemento radice, il primo metodo è consigliabile. Il secondo metodo è preferibile se si vuole accedere anche ad altri nodi (tuttavia si dovrà un po' modificare il codice).
Ottenuto un determinato nodo, l'interfaccia `Node` (implementata da `Element`) offre diversi metodi per esplorare altri nodi relativamente al nodo in questione. Per ottenere il nodo *padre* del nodo che si sta ispezionando, esiste il metodo `getParentNode()`:

```
Node parent = node.getParentNode();
```

Per ottenere la lista dei nodi *figlio* di un nodo, esiste il metodo `getChildNodes()` (come già visto negli esempi precedenti):

```
NodeList children = node.getChildNodes();
```

Ma è anche possibile ottenere solo il primo o solo l'ultimo dei nodi figlio, grazie ai metodi `getFirstChild()` e `getLastChild()` utilizzati nel modo seguente:

```
Node child = node.getFirstChild();
```

e

```
Node child = node.getLastChild();
```

I metodi `getNextSibling()` e `getPreviousSibling()` permettono invece di accedere ai nodi che si trovano allo stesso livello di ramificazione (successivo e precedente).
Considerando il seguente codice XML:

```
<eje>
  <preferences>
    <tab-spaces>4</tab-spaces>
    <work-dir>/home/cdsc/projects/EJE</work-dir>
    <lang>it_IT</lang>
    <class-wizard>on</class-wizard>
    <eje-style>default</eje-style>
    <braces-style>C</braces-style>
    <prologue>Written by me</prologue>
    <colors>
      <keyword>blue</keyword>
      <common>dark_grey</common>
      <string>green</string>
      <char>dark_red</char>
      <number>dark_red</number>
      <operator>light_grey</operator>
    </colors>
  </preferences>
</eje>
```

possiamo osservare che i tag `tab-spaces` e `work-dir` sono sullo stesso livello di ramificazione, così come i tag `keyword` e `common`.
Con il seguente codice si accede al nodo successivo che si trova allo stesso livello di ramificazione:

```
Node sibling = node.getNextSibling();
```

Col seguente codice invece, si accede al nodo fratello precedente:

```
Node sibling = node.getPreviousSibling();
```

> **I metodi** `getFirstChild()`, `getLastChild()`, `getNextSibling()` **e** `getPreviousSibling()` **restituiranno** `null` **nel caso in cui non trovino quanto richiesto.**

Purtroppo la libreria DOM a volte non è molto intuitiva, e bisogna utilizzare i pochi metodi a disposizione per poter accedere a determinati nodi. Per esempio, con i prossimi due frammenti di codice si accede rispettivamente al primo e all'ultimo nodo allo stesso livello di ramificazione:

```
Node sibling = node.getParentNode().getFirstChild();
```

e

```
Node sibling = node.getParentNode().getLastChild();
```

N.1.1.4 Modifica di un documento XML

Il codice seguente crea un documento XML da zero e aggiunge vari tipi di nodi, sfruttando diversi metodi:

```
1  DocumentBuilderFactory factory =
2    DocumentBuilderFactory.newInstance();
3  factory.setValidating(false);
4  Document doc = factory.newDocument();
5  Element root = doc.createElement("prova");
6  doc.appendChild(root);
7  Comment comment = doc.createComment("prova commento");
8  doc.insertBefore(comment, root);
9  Text text = doc.createTextNode("prova testo");
10 root.appendChild(text);
```

Con le prime quattro righe creiamo un documento XML vuoto. Con le righe 5 e 6, prima creiamo l'elemento root che chiamiamo `prova`, e poi lo aggiungiamo al documento con il metodo `appendChild()`. Con le righe 7 e 8 invece creiamo un commento che poi anteponiamo al documento root. Con le righe 9 e 10 infine, viene creato e aggiunto del testo all'unico elemento del documento con il metodo `appendChild()`.

Come già asserito in precedenza, è possibile notare come anche il testo di un nodo sia considerato un nodo. Infatti l'interfaccia `Text` implementa `Node`.

Il documento finale sarà il seguente:

```
<?xml version="1.0" encoding="UTF-8"?>
<!—prova commento-->
<prova>
Prova testo
</prova>
```

Se come testo inserissimo caratteri che per XML sono considerati speciali, come "<" o ">", sarebbero automaticamente convertiti dall'XML writer che si occupa di creare il documento nelle relative entità: rispettivamente in "`&LT;`" e "`&GT;`".

Per rimuovere un nodo è possibile utilizzare il metodo `removeChild()` nel seguente modo:

```
NodeList list = doc.getElementsByTagName("prova");
Element element = (Element)list.item(0);
element.getParentNode().removeChild(element);
```

Nell'esempio abbiamo dapprima individuato un nodo specifico all'interno del documento, grazie al metodo `getElementsByTagName()` su cui è stato chiamato il metodo `item(0)`. Infatti, `getElementsByTagName()` restituisce un oggetto `NodeList`, che contiene tutti i tag con il nome specificato. Con `item(0)` viene restituito il primo della lista. Infine, per rimuovere il nodo individuato, abbiamo dapprima dovuto riprendere il nodo padre, per poi cancellarne il figlio con il metodo `removeChild()`.

Rimuovere un nodo non significa rimuovere i suoi sottonodi, e quindi neanche gli eventuali text node.

N.1.2 Simple API for XML (SAX)

La libreria **SAX**, acronimo di **Simple API for XML**, a differenza di DOM supporta l'analisi di un documento XML basata su eventi, e non può modificare l'XML analizzato. Inoltre SAX non crea un'immagine in memoria del testo XML, e quindi consuma meno memoria. È definita per lo più dalle interfacce del package `org.xml.sax` e dei suoi sottopackage. Nel seguente esempio utilizziamo un parsing SAX per esplorare un file XML, ottenendo la stampa di tutti i tag del documento:

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class SAXExample {

    public static void main(String[] args) {
        DefaultHandler myHandler = new MyHandler();
        try {
            SAXParserFactory factory = SAXParserFactory.newInstance();
            factory.setValidating(false);
            factory.newSAXParser().parse(new File("eje.xml"), myHandler);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static class MyHandler extends DefaultHandler {
        public void startDocument() {
            System.out.println("---Inizio parsing---");
        }

        public void startElement(String uri, String localName,
          String qName, Attributes attributes) {
            System.out.println("<"+qName+ ">");
        }

        public void endElement(String uri, String localName,
          String qName) {
            System.out.println("</"+ qName+ ">");
        }

        public void endDocument() {
            System.out.println("---Fine parsing---");
        }
    }
}
```

SAX si basa sul concetto di gestore di eventi (handler). Quando viene eseguito un

parsing di un documento tramite SAX, la lettura di ogni nodo rappresenta un evento da gestire. Analizzando l'esempio, concentriamoci dapprima sulla classe innestata `MyHandler`. Come è possibile notare, `MyHandler` è sottoclasse di `DefaultHandler` e ne ridefinisce alcuni metodi. Questi, come già si può intuire dai loro nomi, vengono chiamati in base agli eventi che rappresentano. Infatti il corpo del metodo `main()` è piuttosto semplice. Si istanzia prima `MyHandler`, poi in un blocco `try-catch` viene istanziato un oggetto `factory` di tipo `SAXParserFactory`, cui viene imposto di ignorare una eventuale validazione del documento con il metodo `setValidating()`. Infine viene eseguito il parsing del file eje.xml su un nuovo oggetto `SAXParser`, specificando come gestore di eventi l'oggetto `myHandler`. Da questo punto in poi sarà la stessa Java Virtual Machine ad invocare i metodi della classe `MyHandler` sull'oggetto `myHandler`, in base agli eventi scatenati dalla lettura sequenziale del file XML.
Considerando, come input del codice precedente, il seguente file eje.xml:

```
<eje>
  <preferences>
    <tab-spaces>4</tab-spaces>
    <work-dir>/home/cdsc/projects/EJE</work-dir>
    <lang>it_IT</lang>
    <class-wizard>on</class-wizard>
    <eje-style>default</eje-style>
    <braces-style>C</braces-style>
    <prologue>Written by me</prologue>
    <colors>
      <keyword>blue</keyword>
      <common>dark_grey</common>
      <string>green</string>
      <char>dark_red</char>
      <number>dark_red</number>
      <operator>light_grey</operator>
    </colors>
  </preferences>
</eje>
```

l'output stamperà tutti i tag trovati come di seguito:

```
---Inizio parsing---
<eje>
<preferences>
<tab-spaces>
</tab-spaces>
<work-dir>
</work-dir>
<lang>
</lang>
```

```
<class-wizard>
</class-wizard>
<eje-style>
</eje-style>
<braces-style>
</braces-style>
<prologue>
</prologue>
<colors>
<keyword>
</keyword>
<common>
</common>
<string>
</string>
<char>
</char>
<number>
</number>
<operator>
</operator>
</colors>
</preferences>
</eje>
---Fine parsing---
```

N.1.3 Streaming API for XML (StAX)

La libreria **StAX**, acronimo di **Streaming API for XML** supporta un approccio che si pone a metà strada tra DOM e SAX. Il punto di partenza dell'analisi di un testo XML, consiste in un cursore che si posiziona su un tag nel documento. Il cursore muove il cursore (solo) in avanti e legge le informazioni che interessano al programma. Può sembrare simile a SAX, che però si basa su eventi che *passano* le informazioni all'applicazione. Con StAX invece le informazioni vengono richieste dal programma stesso. È definita soprattutto nei tipi del package `javax.xml.stream` e dei suoi sottopackage. Consideriamo il seguente XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<emails>
  <email>
    <from>claudio@claudiodesio.com</from>
    <recipients>daxixlx.sxpxrx@gmail.com</recipients>
    <subject>Nuovo progetto</subject>
    <body>A che punto siamo?</body>
  </email>
  <email>
    <from>daxixlx.sxpxrx@gmail.com</from>
    <recipients>claudio@claudiodesio.com</recipients>
```

```
    <subject>Re: Nuovo progetto</subject>
    <body>Ho quasi finito!</body>
  </email>
  <email>
    <from>claudio@claudiodesio.com</from>
    <recipients>daxixlx.sxpxrx@gmail.com</recipients>
    <subject>Nuovo progetto</subject>
    <body>Ok, concludiamo! Ciao!</body>
  </email>
</emails>
```

che astrae una serie di email, per ognuna delle quali viene specificato il mittente (tag `from`), i destinatari (tag `recipients`), l'oggetto della mail (tag `subject`) e il corpo della mail (tag `body`). StAX richiede di creare una classe che replichi la struttura del testo XML:

```
public class EMail {

    private String from;
    private String recipients;
    private String subject;
    private String body;

    public String getFrom() {
        return from;
    }

    public void setFrom(String from) {
        this.from = from;
    }

    public String getRecipients() {
        return recipients;
    }

    public void setRecipients(String recipients) {
        this.recipients = recipients;
    }

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public String getBody() {
        return body;
    }
```

```
    public void setBody(String body) {
        this.body = body;
    }

    @Override
    public String toString() {
        return "\nEMail da " + from + " a " + recipients
          + "\noggetto: " + subject + "\n" + body;
    }
}
```

Con il seguente codice andiamo a stampare i contenuti delle email:

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.List;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.events.EndElement;
import javax.xml.stream.events.StartElement;
import javax.xml.stream.events.XMLEvent;

public class StAXExample {

    // dichiariamo costanti di utilità
    private static final String EMAIL = "email";
    private static final String FROM = "from";
    private static final String RECIPIENTS = "recipients";
    private static final String SUBJECT = "subject";
    private static final String BODY = "body";

    public List<EMail> readConfig(String xmlFile) {
        List<EMail> eMails = new ArrayList<>();
        try {
            // creaiamo un XMLInputFactory...
            XMLInputFactory inputFactory = XMLInputFactory.newInstance();
            // ...e poi un XMLEventReader
            InputStream in = new FileInputStream(xmlFile);
            XMLEventReader xmlEventReader =
              inputFactory.createXMLEventReader(in);
            EMail eMail = null;
            // leggiamo il file xml
            while (xmlEventReader.hasNext()) {
                XMLEvent xmlEvent = xmlEventReader.nextEvent();
                if (xmlEvent.isStartElement()) {
                    StartElement startElement = xmlEvent.asStartElement();
                    String localPart = startElement.getName().getLocalPart();
```

```
                    // a seconda del tag che stiamo leggendo creiamo un
                    // oggetto Email e settiamo le sue variabili d'istanza
                    switch (localPart) {
                        case EMAIL:
                            eMail = new EMail();
                            break;
                        case FROM:
                            xmlEvent = xmlEventReader.nextEvent();
                            eMail.setFrom(xmlEvent.asCharacters().getData());
                            break;
                        case RECIPIENTS:
                            xmlEvent = xmlEventReader.nextEvent();
                            eMail.setRecipients(
                                  xmlEvent.asCharacters().getData());
                            break;
                        case SUBJECT:
                            xmlEvent = xmlEventReader.nextEvent();
                            eMail.setSubject(
                              xmlEvent.asCharacters().getData());
                            break;
                        case BODY:
                            xmlEvent = xmlEventReader.nextEvent();
                            eMail.setBody(xmlEvent.asCharacters().getData());
                            break;
                        default:
                            break;
                    }
                }
                // se siamo arrivati al tag di chiusura dell'email aggiungiamo
                // l'eMail all'arraylist eMails
                if (xmlEvent.isEndElement()) {
                    EndElement endElement = xmlEvent.asEndElement();
                    if (endElement.getName().getLocalPart().equals(EMAIL)) {
                        eMails.add(eMail);
                    }
                }
            }
        } catch (FileNotFoundException | XMLStreamException e) {
            e.printStackTrace();
        }
        return eMails;
    }

    public static void main(String args[]) {
        StAXExample stAXExample = new StAXExample();
        List<EMail> eMails = stAXExample.readConfig("emails.xml");
        for (EMail eMail : eMails) {
            System.out.println(eMail);
        }
    }
}
```

Il codice è spiegato all'interno dei commenti. Si noti che l'oggetto che rappresenta il cursore che avanza tra i tag del codice XML, è di tipo `XMLEventReader`.
Se volessimo invece modificare il testo XML, allora dovremmo utilizzare un oggetto di tipo `XMLEventWriter` come è possibile verificare nel seguente esempio auto esplicativo (vedi commenti):

```
import java.io.FileOutputStream;
import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.XMLEventWriter;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.events.Characters;
import javax.xml.stream.events.EndElement;
import javax.xml.stream.events.StartDocument;
import javax.xml.stream.events.StartElement;

public class StAXExampleWriter {

    public void salvaFileXML(String fileXML) throws Exception {
        // creiamo un XMLOutputFactory
        XMLOutputFactory xmlOutputFactory = XMLOutputFactory.newInstance();
        // crea un XMLEventWriter
        XMLEventWriter xmlEventWriter = xmlOutputFactory
          .createXMLEventWriter(new FileOutputStream(fileXML));
        // creiamo un XMLEventFactory
        XMLEventFactory xmlEventFactory = XMLEventFactory.newInstance();
        //XMLEvent end = xmlEventFactory.createDTD("\n");
        // creiamo e aggiungiamo a xmlEventWriter un inizio di documento
        StartDocument startDocument = xmlEventFactory.createStartDocument();
        xmlEventWriter.add(startDocument);
        Characters acapo = xmlEventFactory.createCharacters("\n");
        xmlEventWriter.add(acapo);
        // creiamo e aggiungiamo a xmlEventWriter il tag 'email' iniziale
        StartElement startElement =
          xmlEventFactory.createStartElement("", "", "utente");
        xmlEventWriter.add(startElement);
        xmlEventWriter.add(acapo);
        //xmlEventWriter.add(end);
        // Write the different nodes
        creaTag(xmlEventWriter, "nome", "Claudio");
        creaTag(xmlEventWriter, "cognome", "De Sio Cesari");
        xmlEventWriter.add(xmlEventFactory.createEndElement("", "", "utente"));
        //xmlEventWriter.add(end);
        xmlEventWriter.add(xmlEventFactory.createEndDocument());
        xmlEventWriter.close();
    }

    // crea un tag innestato con indentazione
    private void creaTag(XMLEventWriter xmlEventWriter,
```

```
        String nomeTag, String contenutoTag) throws XMLStreamException {
          XMLEventFactory xmlEventFactory = XMLEventFactory.newInstance();
          // create il tag di apertura e aggiungiamolo a xmlEventWriter
          StartElement tagDiApertura =
            xmlEventFactory.createStartElement("", "", nomeTag);
          Characters acapo = xmlEventFactory.createCharacters("\n");
          Characters spazi = xmlEventFactory.createCharacters("  ");
          xmlEventWriter.add(spazi);
          xmlEventWriter.add(tagDiApertura);
          // creiamo il contenuto del tag e aggiungiamolo a xmlEventWriter
          Characters characters =
            xmlEventFactory.createCharacters(contenutoTag);
          xmlEventWriter.add(characters);
          // creiamo il tag di chiusura
          EndElement tagDiChiusura =
            xmlEventFactory.createEndElement("", "", nomeTag);
          xmlEventWriter.add(tagDiChiusura);
          xmlEventWriter.add(spazi);
          xmlEventWriter.add(acapo);
      }

      public static void main(String[] args) {
          StAXExampleWriter stAXExampleWriter = new StAXExampleWriter();
          try {
              stAXExampleWriter.salvaFileXML("testScritturaConStAx.xml");
              System.out.println("File salvato!");
          } catch (Exception e) {
              e.printStackTrace();
          }
      }
  }
```

L'output di questo programma sarà la creazione del file testScritturaConStAx.xml, che conterrà il seguente testo:

```
<?xml version="1.0"?>
<utente>
  <nome>Claudio</nome>
  <cognome>De Sio Cesari</cognome>
</utente>
```

N.1.4 XPATH

Come è facile immaginare, se il documento da analizzare ha una struttura molto ramificata, non sarà sempre agevole riuscire ad analizzare uno specifico nodo, visto che la ricerca potrebbe essere anche molto complessa. Per semplificare la ricerca dei nodi XML esiste un linguaggio appositamente creato che si chiama **XPath** (le cui specifiche si trovano all'indirizzo

https://www.w3.org/TR/xpath-3). Si tratta di uno dei linguaggi definiti dalla tecnologia **XSLT** insieme a **XSL** e **XSL-FO**, allo scopo di trasformare i documenti da XML in altri formati. Ma XPath ha trovato applicazione anche in altre tecnologie XML-based, ed è relativamente conosciuto da molti sviluppatori. Java supporta XPath ufficialmente solo dalla versione 5 mediante la definizione del package `javax.xml.xpath`, appartenente al modulo `java.xml`. Il linguaggio XPATH permette di raggiungere un determinato nodo (o attributo o testo etc.) mediante una sintassi semplice, senza dover obbligatoriamente esplorare l'intero albero DOM del documento XML. Per esempio, anteporre due slash (simbolo "/") prima del nome del nodo all'interno di un'espressione XPATH significa voler cercare il primo nodo con quell'identificatore, indipendentemente dalla sua posizione nel documento XML. Posporre poi parentesi quadre che circondano un numero intero `i` al nome di un nodo significa volere l'i-esimo nodo con quell'identificatore. Il simbolo di chiocciola invece serve per specificare gli attributi dei nodi. Per esempio:

```
//percorso del file XML da analizzare
final static String FILE_XML = "resources/file.xml";
//istanza dell'oggetto XPath
XPath xpath = XPathFactory.newInstance().newXPath();
//creazione di un oggetto InputSource, un che leggerà il file XML
InputSource inputSource = new InputSource(FILE_XML);
for (int i = 1; true; i++) {
    String expression = String.format("//http-request[%d]", i);
    Node node = null;
    try {
        node = (Node) xpath.evaluate(expression, inputSource,
                                     XPathConstants.NODE);
        if (node == null) {
            break;
        }
        String requestURI = (String) xpath.evaluate("@requesturi", node);
        System.out.println("node " + node);
        System.out.println("requestURI " + requestURI);
    } catch (XPathExpressionException exc) {
        exc.printStackTrace();
    }
}
```

Questo codice legge un file XML andandone a stampare tutti i nodi `http-request` e i relativi attributi `requesturi`. Ma le potenzialità di XPATH non si fermano qui. La seguente tabella mostra una serie di espressioni XPATH e il relativo significato:

Espressione	Significato
`expression = "/*";`	Elemento root senza specificare il nome
`expression = "/root";`	Elemento root specificando il nome
`expression = "/root/*";`	Tutti gli elementi subito sotto root
`expression = "/root/node1";`	Tutti gli elementi `node1` subito sotto root
`expression = "//node1";`	Tutti gli elementi `node1` nel documento
`expression = "//node1[4]";`	Il quarto elemento `node1` trovato nel documento
`expression = "//*[name()!='node1'";`	Tutti gli elementi che non siano `node1`
`expression = "//node1/node2";`	Tutti i `node2` che sono figli di `node1`
`expression = "//*[not(node1)]";`	Tutti gli elementi il cui figlio non è `node1`
`expression = "//*[*]";`	Tutti gli elementi con almeno un elemento figlio
`expression = "//*[count(node1) > 3]";`	Tutti gli elementi con almeno tre `node1` figli
`expression = "/*/*/node1";`	Tutti i `node1` al terzo livello

N.1.5 Trasformazioni XSLT

Il package `javax.xml.transform` e i suoi sottopackage consentono di utilizzare le trasformazioni dei documenti XML in base alla tecnologia XSLT (per informazioni su XSLT: http://www.w3c.org). Questo package definisce due interfacce chiamate `Source` (sorgente da trasformare) e `Result` (risultato della trasformazione) che sono utilizzate per le trasformazioni, la cui implementazione richiede però che siano usate delle classi concrete.

Esistono tre implementazioni di coppie `Source` - `Result`:

- `StreamSource` e `StreamResult`;
- `SAXSource` e `SAXResult`;
- `DOMSource` e `DOMResult`.

`StreamSource` e `StreamResult` essenzialmente servono per avere rispettivamente in input e output flussi di dati, per esempio file. Quindi, se volessimo scrivere un documento XML in un file, potremmo utilizzare il seguente codice:

```
try {
    Source source = new DOMSource(doc);
    File file = new File("fileName.xml");
    Result result = new StreamResult(file);
    Transformer transformer =
      TransformerFactory.newInstance().newTransformer();
    transformer.transform(source, result);
} catch (Exception e) {
    e.printStackTrace();
}
```

Dando per scontato che `doc` sia un documento DOM, come sorgente abbiamo utilizzato un oggetto di tipo `DOMSource`. Siccome vogliamo scrivere in un file, abbiamo utilizzato come oggetto di output uno `StreamResult`. Il metodo statico `newInstance()` istanzierà un oggetto di tipo `TransformerFactory`, che a sua volta istanzierà un oggetto di tipo `Transformer`. L'oggetto `transformer`, grazie al metodo `transform()`, trasformerà il contenuto del documento DOM nel file.

Gli oggetti di tipo `Source` e `Result` possono essere utilizzati una sola volta dopodiché bisogna istanziarne altri.

Inoltre, la classe `Transformer` definisce il metodo `setOutputProperty()` che si può sfruttare per garantirsi output personalizzati (consultare la documentazione per dettagli). Per esempio, se il seguente codice fosse eseguito prima del metodo `transform()` dell'esempio precedente, provocherebbe la scrittura nel file XML del solo testo e non dei tag del documento:

```
transformer.setOutputProperty(OutputKeys.METHOD, "text");
```

Ma con XSLT è possibile fare molto di più. Con il seguente codice è possibile ottenere un transformer basato su un file XSL per una trasformazione del documento

DOM:

```
TransformerFactory tf = TransformerFactory.newInstance();
Templates template =
  tf.newTemplates(new StreamSource(new FileInputStream("fileName.xsl")));
Transformer transformer = template.newTransformer();
```

Il resto del codice rimane identico a quello dell'esempio precedente.

N.1.6 Java API for XML Bindings (JAXB)

La libreria **JAXB**, acronimo di **Java Architecture for XML Bindings**, si pone come alternativa a JAXP per l'analisi di testo XML, ed è definita soprattutto tramite i tipi definiti in `javax.xml.bind` e i suoi sottopackage. L'approccio di JAXB però è diverso visto che consiste nel mappare il contenuto XML all'interno di un oggetto Java e viceversa. Supporta l'analisi e la scrittura di testo XML. È definito tramite una specifica e quindi possono esistere diverse implementazioni. Nel seguente esempio mostriamo come si utilizza JAXB, mostrando le sue caratteristiche di base. Partendo da classi Java scriviamo sull'hard disk un file XML. Poi lo rileggiamo e ne stampiamo i contenuti tramite oggetti Java. Dovendo utilizzare delle annotazioni che si trovano nel package `javax.xml.bind.annotation`, che appartiene al modulo `java.xml.bind`, iniziamo a creare un modulo che legge il modulo `java.xml.bind`, e apre alla lettura tramite reflection allo stesso modulo il package in cui definiremo le nostre classi (cfr. capitolo 19). A questo modulo per convenzione diamo lo stesso nome del package delle nostre classi, ovvero `com.claudiodesio.appendici.n`:

```
module com.claudiodesio.appendici.n {
    requires java.xml.bind;
    opens com.claudiodesio.appendici.n to java.xml.bind;
}
```

Possiamo quindi iniziamo a creare le classi Java che vogliamo salvare sotto forma XML. La prima classe si chiama `Song` ed astrae un oggetto di tipo canzone:

```
package com.claudiodesio.appendici.n;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "song")
// opzionalmente possiamo specificare l'ordine in cui i campi verranno scritti
// con l'annotazione XmlType che specifica l'elemento propOrder:
@XmlType(propOrder = {"title", "artist", "album", "year"})
public class Song {
```

```
    private String title;
    private String artist;
    private String album;
    private String year;

    // possiamo cambiare il nome nell'output XML aggiungendo l'annotazione
    // XmlElement er specificando un nome diverso
    @XmlElement(name = "titolo")
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getArtist() {
        return artist;
    }

    public void setArtist(String artist) {
        this.artist = artist;
    }

    public String getAlbum() {
        return album;
    }

    public void setAlbum(String album) {
        this.album = album;
    }

    public String getYear() {
        return year;
    }

    public void setYear(String year) {
        this.year = year;
    }

    @Override
    public String toString() {
        return "Canzone: " + "title=" + title + ", artist=" + artist
          + ", album=" + album + ", year=" + year;
    }
}
```

Si tratta di una classe molto semplice con quattro variabili d'istanza incapsulate. Si noti che abbiamo utilizzato tre annotazioni. L'annotazione `XmlRootElement`

ci è servita per dichiarare che `song` è l'elemento radice. L'elemento `propOrder` dell'annotazione `XMLType` permette di specificare l'ordine in cui verranno salvati i tag XML. Infine l'annotazione `XMLElement` permette di specificare un nome alternativo per il tag XML che sarà creato relativamente alla proprietà annotata. Nel nostro caso verrà creato il tag `titolo` in luogo del tag `title`. Creiamo ora una classe `Playlist`, che contiene oggetti di tipo `Song`:

```
package com.claudiodesio.appendici.n;

import java.util.ArrayList;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Playlist {

    private ArrayList<Song> songs;
    private String author;

    public ArrayList<Song> getSongs() {
        return songs;
    }

    public void setSongs(ArrayList<Song> songs) {
        this.songs = songs;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```

Anche questa classe è molto semplice. Infine creiamo la classe principale opportunamente commentata:

```
package com.claudiodesio.appendici.n;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.PropertyException;
```

```
import javax.xml.bind.Unmarshaller;

public class JAXBExample {

    private static final String PLAYLIST_FILE = "playlist_con_jaxb.xml";

    public static void main(String[] args) throws IOException, JAXBException {
        // creaimo una playlist
        Playlist playlist = getPlaylist();
        // creazione di un oggetto JAXBContext
        JAXBContext context = JAXBContext.newInstance(Playlist.class);
        // salviamo la playlist in un file XML
        salvaPlaylistSuXml(context, playlist);
        // rileggiamo il contenuto della playlist direttamente dal file
        leggiPlaylistDaFileXML(context);
    }

    private static void leggiPlaylistDaFileXML(JAXBContext context)
      throws FileNotFoundException, JAXBException {
        // creaiamo un oggetto unmarshaller per leggere dal file
        Unmarshaller unmarshaller = context.createUnmarshaller();
        // leggiamo la playlista dal file XML
        Playlist playlistLetta =
          (Playlist) unmarshaller.unmarshal(new FileReader(PLAYLIST_FILE));
        // recuperiamo le canzoni e stampiamo
        ArrayList<Song> songsLette = playlistLetta.getSongs();
        System.out.println("Contenuto della playlist letta dal file XML: ");
        for (Song song : songsLette) {
            System.out.println(song);
        }
    }

    private static void salvaPlaylistSuXml(JAXBContext context,
      Playlist playlist) throws JAXBException, PropertyException {
        // creiamo un oggetto Marshaller cper scrivere i un file.
        Marshaller marshaller = context.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
        // scriviamo la playlist nel file XML
        marshaller.marshal(playlist, new File(PLAYLIST_FILE));
        System.out.println("Salvato il seguente file XML:");
        // istruzione che scrive il file sull'oggetto System.out
        marshaller.marshal(playlist, System.out);
    }

    private static Playlist getPlaylist() {
        ArrayList<Song> songs = getListaDiCanzoni();
        // creazione di un oggetto Playlist
        Playlist playlist = new Playlist();
        playlist.setAuthor("Claudio De Sio");
        playlist.setSongs(songs);
        return playlist;
```

```
    }

    private static ArrayList<Song> getListaDiCanzoni() {
        // creaiamo un arraylist di oggetti song
        ArrayList<Song> songs = new ArrayList<>();
        //creazione della prima canzone
        Song song1 = new Song();
        song1.setTitle("The road of Bones");
        song1.setArtist("IQ");
        song1.setAlbum("The road of Bones");
        song1.setYear("2014");
        songs.add(song1);
        // creazione della seconda canzone
        Song song2 = new Song();
        song2.setTitle("In the passing light of day");
        song2.setArtist("Pain of salvation");
        song2.setAlbum("In the passing light of day");
        song2.setYear("2017");
        songs.add(song2);
        return songs;
    }
}
```

Per compilare i nostri file all'interno del modulo abbiamo creato, in una cartella di base, due sottocartelle `src` e `mods`. In `src` abbiamo creato una cartella chiamata `com.claudiodesio.appendici.n` che rappresenta il modulo dichiarato. All'interno di tale cartella vengono posti i sorgenti all'interno dell'alberatura di package corrispondente alla dichiarazione dei package dei sorgenti stessi.

Abbiamo già impostato i nostri progetti nello stesso modo nel capitolo 19.

Per compilare, ci dobbiamo posizionare nella cartella radice ed eseguire il seguente comando dalla prompt DOS:

```
javac -d mods --module-source-path src
  src/com.claudiodesio.appendici.n/module-info.java src/
  com.claudiodesio.appendici.n/com/claudiodesio/appendici/n/*
```

Il modulo è stato deprecato, quindi l'output della compilazione darà luogo ad un warning:

```
module-info.java:2: warning: [removal] module java.xml.bind has been
  deprecated and marked for removal
    requires java.xml.bind;
```

```
                    ^
1 warning
```

Questo per ora non pregiudica il cattivo funzionamento dell'applicazione, ma in futuro questo modulo scomparirà per essere sostituito. Per eseguire l'applicazione invece bisogna esegurie il seguente comando:

```
java --module-path mods -m
  com.claudiodesio.appendici.n/com.claudiodesio.appendici.n.JAXBExample
```

Trovate tutti i sorgenti e gli script di compilazione tra il codice on line nella cartella relativa a quest'appendice.

L'output di questo programma è il seguente:

```
Salvato il seguente file XML:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<playlist>
    <author>Claudio De Sio</author>
    <songs>
        <titolo>The road of Bones</titolo>
        <artist>IQ</artist>
        <album>The road of Bones</album>
        <year>2014</year>
    </songs>
    <songs>
        <titolo>In the passing light of day</titolo>
        <artist>Pain of salvation</artist>
        <album>In the passing light of day</album>
        <year>2017</year>
    </songs>
</playlist>
Contenuto della playlist letta dal file XML:
Canzone: title=The road of Bones, artist=IQ, album=The road of Bones,
  year=2014
Canzone: title=In the passing light of day, artist=Pain of salvation,
  album=In the passing light of day, year=2017
```

Appendice O

Prima del Framework Collections

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

- ✔ Essere in grado di utilizzare le collezioni storiche esistenti precedentemente alla realizzazione del framework "Collections" (unità O.1, O.2, O.3).
- ✔ Essere consapevoli della difficoltà di implementare collezioni generiche (unità O.4, O.5).

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 nella versione 1.0 esistevano già 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`.

O.1 La classe `Hashtable`

La classe `Hashtable` è una tipica implementazione di una mappa. Questa classe permette di associare ad ogni elemento della collezione una chiave univoca. La chiave e l'elemento associato sono definiti rispettivamente dal primo e dal secondo tipo parametro dello `Hashtable`. Si aggiungono elementi mediante il metodo `put()` e si recuperano tramite il metodo `get()`. In particolare il metodo `get()` consente un recupero molto performante dell'elemento della collezione specificando la chiave. Per esempio, il seguente codice:

```
HashMap<Integer,String> table = new HashMap<>();
```

```
table.put(2006,"Simone");
table.put(2004, "Andrea");
System.out.println(table.get(2004));
System.out.println(table.get(2006));
```

produrrà l'output:

```
Andrea
Simone
```

Non sono ammesse chiavi duplicate né elementi `null`. Inoltre un oggetto `Hashtable` è thread-safe di default, e questo lo penalizza dal punto di vista delle performance, ed è questa una delle ragioni essenziali per cui è molto meno utilizzata della sua omologa `HashMap`.

0.2 La classe `Vector`

È stata per qualche anno la collezione più usata del linguaggio, e ancora oggi abbiamo nella libreria diversi esempi di utilizzo di `Vector`. Oggi è stata soppiantata da `ArrayList` che per molti versi è un suo clone non thread-safe. In generale quindi la classe `Vector` offre prestazioni inferiori rispetto ad un `ArrayList`, essendo sincronizzata. Per questo utilizza due parametri per configurare l'ottimizzazione delle prestazioni: la capacità iniziale e la capacità d'incremento. Per quanto riguarda la capacità iniziale vale quanto detto per `ArrayList` (cfr. capitolo 17). La capacità d'incremento (specificabile tramite un costruttore) permette di stabilire di quanti posti si deve incrementare la capacità del vettore ogniqualvolta si aggiunga un elemento che sfora il numero di posizioni disponibili. Se per esempio istanziamo un `Vector` nel seguente modo:

```
Vector<String> v = new Vector<>(10, 10);
```

dove il primo parametro è la capacità iniziale e il secondo la capacità di incremento, quando aggiungeremo l'undicesimo elemento la capacità del vettore sarà reimpostata a `20`. Quando aggiungeremo il ventunesimo elemento la capacità del vettore sarà reimpostata a `30` e così via. Il seguente codice:

```
Vector<String> list = new Vector<>(10,10);
for (int i = 0; i < 11; i++) {
    list.add("1");
}
System.out.println("Capacità = " + list.capacity());
for (int i = 0; i < 11; i++) {
    list.add("1");
}
System.out.println("Capacità = " + list.capacity());
```

produrrà il seguente output:

```
Capacità = 20
Capacità = 30
```

Se istanziamo un vettore senza specificare la capacità di incremento, oppure assegnandogli come valore un intero minore o uguale a zero, la capacità sarà raddoppiata ad ogni sforamento. Se quindi istanziamo un `Vector` nel seguente modo:

```
Vector<String> v = new Vector<>();
```

dove non sono state specificate capacità iniziale (che di default viene impostata a `10`) e capacità di incremento (impostata per default a `0`), quando aggiungeremo l'undicesimo elemento la capacità del vettore sarà reimpostata a `20`. Quando aggiungeremo il ventunesimo elemento la capacità del vettore sarà reimpostata a `40` e così via, raddoppiando la capacità del `Vector` tutte le volte che occorre ampliarlo.

O.3 L'interfaccia `Enumeration`

Un'altra interfaccia che bisogna menzionare è `Enumeration`. È molto simile all'`Iterator` (cfr. paragrafo 17.2.2.2) ed ha esattamente la stessa funzionalità. Ci sono solo due differenze tra queste due interfacce:

- i nomi dei metodi (che rimangono però molto simili);
- la capacità di un `Iterator` di rimuovere elementi durante l'iterazione, mediante il metodo `remove()`.

In effetti l'interfaccia `Iterator` è nata solo nella versione 1.2 di Java proprio per sostituire `Enumeration` nel framework Collections. Abbiamo menzionato l'esistenza di quest'ultima perché esistono ancora diversi utilizzi di essa all'interno della libreria standard.

Attenzione a non confondere questa interfaccia con il concetto di enumerazione spiegato nel capitolo 10.

O.4 Implementazione di un tipo `Iterable`

È possibile, e alcune volte auspicabile, creare collezioni personalizzate, magari estendendo una collection già pronta e completa. Per esempio possiamo facilmente estendere la classe `Vector` ed aggiungere nuovi metodi. In tali casi sarà già possibile sfruttare il nostro nuovo tipo all'interno di un ciclo for migliorato. Infatti, la condizione necessaria affinché una classe sia parte di un costrutto `foreach` è che

implementi l'interfaccia `Iterable`. `Vector`, come tutte le altre collection, implementa tale interfaccia.
Implementare l'interfaccia `Iterable`, significa implementare un unico metodo: il metodo `iterator()`. Segue la definizione dell'interfaccia `Iterable`:

```
package java.lang;
public interface Iterable<E> {
    public java.util.Iterator<E> iterator();
}
```

Questo metodo, che dovrebbe già essere familiare al lettore, restituisce un'implementazione dell'interfaccia `Iterator` più volte utilizzata in questo testo. Non è così quindi banale implementare da zero `Iterable`, perché significa anche definire un `Iterator` personalizzato. Non si tratta quindi di definire solo il metodo `iterator()`, ma anche implementare tutti i metodi dell'interfaccia `Iterator`, che è così definita:

```
package java.util;
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void remove();
}
```

0.5 Collection personalizzate

Infine bisogna dire che ogni collection si può estendere per creare collection personalizzate. Se volete estendere qualcosa di meno definito rispetto ad un `Vector` o un `ArrayList`, avete a disposizione una serie di classi astratte:

- `AbstractCollection`: implementazione minimale di `Collection`. Bisogna definire i metodi `iterator()` e `size()`.
- `AbstractSet`: implementazione di un `Set`. Bisogna definire i metodi `iterator()` e `size()`.
- `AbstractList`: implementazione minimale di una `List`. Bisogna definire i metodi `get(int)` e, opzionalmente, `set(int)`, `remove(int)`, `add(int)` e `size()`.
- `AbstractSequentialList`: implementazione di una `LinkedList`. Bisogna definire i metodi `iterator()` e `size()`.
- `AbstractMap`: implementazione di una `Map`. Bisogna definire il metodo `entrySet()` (che è solitamente implementato con la classe `AbstractSet`) e se la mappa deve essere modificabile occorre definire anche il metodo `put()`.

Appendice P

Introduzione al networking

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

- ✔ Avere un'infarinatura di base dei concetti fondamentali del networking (unità P.1).
- ✔ Saper creare una semplice coppia di client e server di rete che interagiscano tra loro (unità P.2).

Il networking in Java è un argomento diverso dall'input-output ma dipendente da esso. Come abbiamo già asserito, infatti, è possibile usare come fonte di lettura o come destinazione di scrittura una rete. Fortunatamente il discorso in questo caso è incredibilmente semplice! Infatti per poter creare applicazioni che comunicano in rete, occorrono solo poche righe di codice. Esiste però un package nuovo da importare: `java.net`, appartenente al modulo `java.base`.

P.1 Concetti fondamentali

Iniziamo con il dare qualche informazione di base sui concetti che dovremo trattare. Esistono interi libri che parlano di questi concetti, ma questa non è la sede adatta per poter approfondire troppo il discorso.

P.1.1 Client e Server

Il networking in Java si basa essenzialmente sul concetto di **socket** (che in italiano si traduce come **presa di corrente**). Una socket potrebbe essere definita come il punto terminale di una comunicazione tramite rete. Per comunicare tramite rete occorrono due socket che alternativamente si scambiano informazioni in input e/o in output. L'architettura di rete più utilizzata ai nostri giorni viene detta **client-server** (cfr.

capitolo 5). In questa architettura esistono almeno due programmi (e quindi due socket): appunto un server e un client.
Un **server**, nella sua definizione più generale, è un'applicazione che una volta eseguita si mette in attesa di connessioni da parte di altri programmi (detti appunto client) con cui comunica per fornire loro servizi. Un server è in esecuzione 24 ore su 24.
Un **client** invece è un'applicazione real-time, ovvero ha un ciclo di vita normale, si esegue e si termina senza problemi. Gode della facoltà di potersi connettere ad un server per usufruire dei servizi messi a disposizione da quest'ultimo.

Nella maggior parte dei casi esiste un unico server a disposizione di più client.

Giusto per dare un'idea della vastità dell'utilizzo di questa architettura, basti pensare che il mondo di Internet è basato su client (per esempio browser come Google Chrome e Mozilla Firefox) e server (per esempio Apache Web Server o IIS) e sulla suite di protocolli nota come **TCP/IP** (e più in particolare su **HTTP**). Il nome TCP/IP deriva infatti da due dei più importanti protocolli che definisce: il **TCP** (acronimo di **Transfer Control Protocol**) e **IP** (acronimo di **Internet Protocol**).

P.1.2 Protocolli di rete

L'argomento è molto vasto (ci sono libri da centinaia di pagine che lo trattano), ma in questa sede ci limiteremo a dare solo un'idea di cosa siano i protocolli di rete. Possiamo dire che la suite di protocolli TCP/IP contiene tutti i **protocolli di comunicazione** che normalmente utilizziamo in Internet. Un **protocollo di comunicazione** definisce le regole della comunicazione. Quando navighiamo con il nostro browser, per esempio, utilizziamo (nella stragrande maggior parte dei casi) il protocollo noto come **HTTP** (acronimo di **HyperText Transfer Protocol**). L'HTTP (come tutti i protocolli) ha regole che definiscono non solo i tipi di informazioni che si possono scambiare client e server, ma anche come gestire le connessioni tra essi. Per esempio un client HTTP (il browser) può visitare un certo sito e quindi richiedere una pagina HTML al server del sito in questione. Questo può restituire o meno la pagina richiesta. A questo punto la connessione tra client e server è già chiusa. Alla prossima richiesta del client si aprirà un'altra connessione con il server. Quando invece mandiamo una e-mail utilizziamo un altro protocollo che si chiama **SMTP**, con regole completamente diverse dall'HTTP. Stesso di-

scorso quando riceviamo una e-mail con il protocollo **POP3**, quando scambiamo file con **FTP** o comunichiamo con **Telnet**. Ogni protocollo ha una sua funzione specifica ed una sua particolare politica di gestione delle connessioni.
Un client ed un server comunicano tramite un canale di comunicazione univoco basato essenzialmente su tre concetti: il numero di porta, l'indirizzo IP e il tipo di protocollo.
Per quanto riguarda il **numero di porta** abbiamo uno standard a 16 bit tra cui scegliere (quindi le porte vanno dalla 0 alla 65535). Le prime 1024 dovrebbero essere dedicate ai protocolli standard. Per esempio HTTP agisce solitamente sulla porta 80, FTP sulla 21 e così via. È possibile però utilizzare i vari protocolli su altre porte diverse da quelle di default. Il concetto di porta è virtuale, non fisico.
L'**indirizzo IP** è la chiave per raggiungere una certa macchina che si trova in rete. Ha una struttura costituita da quattro valori interi a otto bit (quindi compresi tra 0 e 255) separati da punti. Per esempio sono indirizzi IP validi 192.168.0.1, 255.255.255.0 e 127.0.0.1 (quest'ultimo è l'indirizzo che individua la macchina locale dove si esegue l'applicazione). Ogni macchina che si connette in rete ha un proprio indirizzo IP. Quando viene eseguito un server questo deve dichiarare su che numero di porta ascolterà le connessioni da parte dei client aprendo un canale di input. Il server inoltre definirà il **protocollo** accettando e rispondendo ai client.

P.2 Esempio

Passando subito alla pratica, creeremo con poche righe di codice un server e un client, che sfruttano un semplicissimo protocollo inventato da noi. La nostra coppia di programmi vuole essere solo un esempio iniziale, ma renderà bene l'idea di cosa significa comunicare in rete con Java.

P.2.1 Server

Scriviamo un semplice server che si mette in ascolto sulla porta 9999 e restituisce, ai client che si connettono, una stringa di saluto per poi interrompere la comunicazione (abbiamo definito noi le regole di questo protocollo):

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

public class SimpleServer {

    public static void main(String args[]) {
        try (ServerSocket serverSocket = new ServerSocket(9999);) {
            System.out.println("Server avviato, in ascolto sulla porta 9999");
            while (true) {
```

```
                try (Socket clientSocket = serverSocket.accept();
                       OutputStream clientOutputStream =
                         clientSocket.getOutputStream();
                       BufferedWriter bufferedWriter = new BufferedWriter(
                         new OutputStreamWriter(clientOutputStream));) {
                    bufferedWriter.write("Ciao client sono il server!");
                    System.out.println("Messaggio spedito a " +
                                       clientSocket.getInetAddress());
                } catch (ConnectException connExc) {
                    System.err.println("Questo client non riesce a " +
                      "connettersi con il server: " +
                      connExc.getMessage());
                } catch (IOException e) {
                    System.err.println("Problema inaspettato: "
                                       + e.getMessage());
                }
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

L'analisi del codice è davvero banale. Istanziando un `ServerSocket` con la porta `9999`, l'applicazione si dichiara server. Poi inizia un ciclo infinito che però si blocca sul metodo `accept()`, il quale mette in stato di "attesa di connessioni" l'applicazione. Una volta ricevuta una connessione da parte di un client, il metodo `accept()` viene eseguito e restituisce il socket che rappresenta il client con tutte le informazioni necessarie. A questo punto serve un canale di output verso il client, che otteniamo semplicemente con il metodo `getOutputStream()` chiamato sull'oggetto socket `clientSocket`. Poi, per comodità, decoriamo questo `OutputStream` con un `BufferedWriter` che, mettendoci a disposizione il metodo `write()`, consente di inviare il messaggio al client in modo molto semplice. Subito dopo stampiamo l'indirizzo del client che riceverà il messaggio. Tutto qui!

Si noti l'utilizzo del costrutto try with resources per tutte le risorse chiudibili.

P.2.2 Client

Scrivere un client che utilizza il server appena descritto è ancora più semplice! Il seguente client, se da riga di comando non è specificato un indirizzo alternativo, suppone che il server sia stato eseguito sulla stessa macchina. Poi, dopo essersi con-

nesso, scrive la frase che riceve dal server. Segue il codice:

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

public class SimpleClient {

    public static void main(String args[]) {
        String host = getServerHost(args);
        try (Socket socketDelServer = new Socket(host, 9999);
               BufferedReader br = new BufferedReader(
                 new InputStreamReader(socketDelServer.getInputStream()));) {
            System.out.println(br.readLine());
        } catch (ConnectException connExc) {
            System.err.println("Non riesco a connettermi al server " +
                               connExc.getMessage());
        } catch (IOException e) {
            System.err.println("Problemi... " + e.getMessage());
        }
    }

    public static String getServerHost(String[] args) {
        String host = "127.0.0.1";
        if (args.length != 0) {
            host = args[0];
        }
        return host;
    }
}
```

Questa applicazione, quando istanzia l'oggetto socket, si dichiara client del server che si trova all'indirizzo `host` e ascolta sulla porta `9999`. Il socket istanziato, quindi, rappresenta il server cui ci si vuole connettere. Poi decora con un `BufferedReader` l'`InputStream` che ricava dal socket mediante il metodo `getInputStream()`. Infine stampa ciò che viene letto dallo stream mediante il metodo `readLine()`.

Si noti anche in questo esempio l'utilizzo del costrutto try with resources per tutte le risorse chiudibili.

Come volevasi dimostrare, scrivere una semplice coppia di client-server è particolarmente facile in Java. Per i programmatori che hanno avuto a che fare con linguaggi come il C++, il package `java.net` potrebbe avere un suono molto dolce. Certamente lo sviluppo di server più complessi richiederebbe molto più impegno, dovendo utilizzare anche il multithreading per la gestione parallela di più

client e magari protocolli ben più sofisticati. Ma ci piace sottolineare che quello appena visto è il modo più complesso di affrontare il networking in Java. Con l'introduzione nella versione 1.4 del package chiamato `java.nio` (ovvero "New Input Output", cfr. capitolo 18) ed i suoi sottopackage, il lavoro degli sviluppatori per creare server e client multithreaded più complessi ed efficienti si è semplificato. I concetti di channel, buffers, selectors e charset permettono di creare con poco sforzo applicazioni molto complesse. Gli oggetti `Selector` possono gestire in maniera semplice oggetti di tipo `SocketChannel` (più di uno alla volta) senza bloccare l'esecuzione del programma. Dalla versione 5 in poi inoltre, esistono nuove classi dette "Executors" (per esempio la classe `ThreadPoolExecutor`) che con poche righe di codice permettono di creare pool di thread per creare applicazioni server con performance notevoli (cfr. paragrafo 15.6.3.3).

Appendice Q

Interfacce grafiche (GUI) con AWT, Applet e Swing

Obiettivi:

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

- ✔ Saper elencare le principali caratteristiche che deve avere una GUI (unità Q.1).
- ✔ Saper descrivere le caratteristiche della libreria AWT (unità Q.2).
- ✔ Saper gestire i principali Layout Manager per costruire GUI complesse (unità Q.3).
- ✔ Saper gestire gli eventi con il modello a delega (unità Q.4).
- ✔ Saper creare semplici applet (unità Q.5).
- ✔ Saper descrivere le caratteristiche più importanti della libreria Swing (unità Q.6).

Nella appendice S è introdotta la tecnologia JavaFX, che Oracle ha oramai dichiarato come il nuovo standard per creare interfacce grafiche. Il vecchio standard Swing non sarà più sviluppato a meno di risoluzioni di bug. Ciononostante esistono ancora migliaia di programmi scritti con i vecchi standard, ed è quindi ancora importante conoscere gli argomenti di seguito presentati. È stato quindi deciso di dedicare questa corposa appendice a quello che ha rappresentato per quasi vent'anni le GUI in Java. Parleremo di AWT, Swing, Applet, HTML ed altri argomenti correlati. Questa appendice si limiterà ad un'introduzione. Infatti ci vorrebbe un intero libro per parlare approfonditamente di interfacce grafiche. Le librerie che vedremo in questa appendice sono tutte appartenenti al modulo `java.desktop`.

Q.1 Introduzione alla Graphical User Interface (GUI)

Oggigiorno le **GUI** (acronimo di **Graphicat User Interface**, in italiano **interfacce grafiche utente**) sono una parte fondamentale delle applicazioni, e questo per varie ragioni. Basti pensare solo al fatto che la maggior parte degli utenti giudica il software principalmente dalla GUI con cui si interfaccia. Questo non vale (o vale meno) per gli utenti che utilizzano gli applicativi più tecnici, come i tool di sviluppo, dove magari si preferiscono le funzionalità. Esistono diversi principi per giudicare un'interfaccia, elenchiamo i più importanti di seguito.

- **Utenza**: è fondamentale tenere ben conto delle tipologie di utenza che usufruiranno della GUI. Per esempio, utenti esperti potrebbero non gradire di dover utilizzare wizard per effettuare alcune procedure. Utenti meno esperti, invece, potrebbero preferire essere guidati.
- **Semplicità**: non bisogna mai dimenticare che l'obiettivo di una GUI è facilitare l'uso dell'applicazione all'utente. Creare GUI semplici, quindi, è una priorità. Bisogna far sì che l'utente non pensi mai di "essersi perso".
- **Usabilità**: una GUI deve offrire un utilizzo semplice ed immediato. Per esempio, implementare scorciatoie con la tastiera potrebbe aiutare molto alcune tipologie di utenti.
- **Estetica**: un ruolo importante lo gioca la piacevolezza dello stile che diamo alla GUI. Essendo soggettiva, non bisognerebbe mai scostarsi troppo dagli standard conosciuti, oppure offrire la possibilità di personalizzazione.
- **Riuso**: una buona GUI, seguendo le regole dell'Object Orientation, dovrebbe anche offrire componenti riutilizzabili. Come già accennato precedentemente però, bisogna scendere a compromessi con l'Object Orientation.
- **Gusti personali e standard**: quando bisogna scegliere come creare una certa interfaccia, non bisogna mai dimenticare che i gusti personali sono sempre trascurabili rispetto agli standard a cui gli utenti sono abituati.
- **Consistenza**: le GUI devono sempre essere consistenti dovunque siano eseguite. Questo significa anche che è fondamentale sempre esporre all'utente informazioni interessanti. Inoltre le differenze che ci sono tra una vista ed un'altra devono essere significative. Infine, qualsiasi sia l'azione (per esempio il ridimensionamento) che l'utente effettua sulla GUI, quest'ultima deve sempre rimanere significativa e utile.
- **Internazionalizzazione**: nel modulo 14 relativo al package `java.util`

abbiamo già parlato di questo argomento. Nel caso di creazione di una GUI, l'internazionalizzazione può diventare molto importante.

- **Model View Controller**: nella creazione delle GUI moderne esiste uno schema architetturale che viene utilizzato molto spesso: il pattern Model View Controller (MVC). Anche se non viene sempre utilizzato nel modo migliore seguendo tutte le sue linee guida, almeno uno dei suoi principi deve assolutamente essere considerato: la separazione dei ruoli. Infatti nell'MVC, si separano tre componenti a seconda del loro ruolo nell'applicazione:

  1. il **model**, che implementa la vera applicazione, ovvero non solo i dati ma anche le funzionalità. Quello che abbiamo studiato sino ad ora serve per creare model (ovvero applicazioni). Si dice che il model, all'interno dell'MVC, implementa la **logica di business** (o **logica applicativa**);
  2. la **view**, che è composta da tutta la parte dell'applicazione con cui si interfaccia l'utente. Questa parte, solitamente costituita da GUI multiple, deve implementare la logica che viene detta **logica di presentazione**, ovvero la logica per organizzare sé stessa. Questo tipo di logica, come vedremo in quest'appendice, è tutt'altro che banale. La view non contiene nessun tipo di funzionalità o dato applicativo, *espone* solamente all'utente le funzionalità dell'applicazione;
  3. il **controller**, che implementa la **logica di controllo**. Questo è il componente più difficile da immaginare in maniera astratta, anche perché non si sente parlare spesso di *logica di controllo*. Giusto per dare un'idea, diciamo solo che questo componente deve avere almeno queste responsabilità: controllare gli input che l'utente immette nella GUI, decidere quale sarà la prossima pagina della view che sarà visualizzata, mappare gli input utente nelle funzionalità del model.

  I vantaggi dell'applicazione dell'MVC sono diversi. Uno di questi, il più evidente, è quello che se deve cambiare l'interfaccia grafica, non deve cambiare l'applicazione. Concludendo, raccomandiamo al lettore quantomeno di non confondere mai il codice che riguarda la logica di business con il codice che riguarda la logica di presentazione.

Per maggiori dettagli ed esempi di codice rimandiamo alla pagina http://www.claudiodesio.com/ooa&d/mvc.htm, interamente dedicata all'MVC. Per la definizione di pattern invece si può consultare l'appendice D.

Alcuni IDE (Integrated Development Editor) per Java, come Netbeans, permettono la creazione di GUI in maniera grafica tramite il trascinamento dei componenti, come avviene con Scene Builder per Java FX ed in altri linguaggi come Visual Basic e Delphi. Questo approccio abbatte i tempi di sviluppo della GUI, ma concede poco spazio alle modifiche a mano e porta a *dimenticare* il riuso. Inoltre, il codice scritto da un IDE non è assolutamente paragonabile a quello scritto da un programmatore. Comunque, in questa appendice gli argomenti saranno presentati come se si dovesse scrivere ogni singola riga.

Quando si dota un programma di una GUI cambia completamente il ciclo di vita del programma stesso. Infatti, mentre tutte le applicazioni senza interfaccia duravano il "tempo di eseguire un `main()`" (e tutti i thread creati), adesso le cose cambiano radicalmente. Una volta che viene visualizzata una GUI, la Java Virtual Machine fa partire un nuovo thread che si chiama "AWT thread", che mantiene sullo schermo la GUI stessa e cattura eventuali eventi su di essa. Quindi un'applicazione che fa uso di GUI, una volta eseguita, rimane in attesa dell'input dell'utente e termina solo in base a un determinato input.

Q.2 Introduzione ad Abstract Window Toolkit (AWT)

AWT è una libreria per creare interfacce grafiche utente sfruttando componenti dipendenti dal sistema operativo. Ciò significa che, eseguendo la stessa applicazione grafica su sistemi operativi differenti, lo stile dei componenti grafici (in inglese detto **Look & Feel**) sarà imposto dal sistema operativo. La figura Q.1 mostra una semplice GUI visualizzata su un vecchio sistema Windows XP.

Figura Q.1 - Una semplice GUI visualizzata su Windows XP.

La GUI in figura Q.1 è stata generata dal seguente codice:

```
import java.awt.*;

public class AWTGUI {

    public static void main(String[] args) {
        Frame frame = new Frame();
        Label l = new Label("AWT", Label.CENTER);
        frame.add(l);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Basta conoscere un po' di inglese per analizzare il codice precedente.
Un'applicazione grafica AWT non si può terminare chiudendo il frame! Infatti, il click sulla X della finestra per la JVM è un *evento da gestire*. Quindi bisognerà studiare l'unità didattica relativa alla gestione degli eventi prima di poter chiudere le applicazioni come siamo abituati. Per adesso bisognerà interrompere il processo in uno dei seguenti modi:

1. se abbiamo eseguito l'applicazione da riga di comando, basterà premere contemporaneamente CTRL-C avendo in primo piano il prompt dei comandi;
2. se abbiamo eseguito l'applicazione utilizzando EJE o un qualsiasi altro IDE si troverà un pulsante o una voce di menu che interrompe il processo. Per EJE trovate il pulsante stop in rosso sulla toolbar. Alternativamente è sufficiente premere il tasto ESC sulla tastiera;
3. in mancanza d'altro si potrà accedere sicuramente in qualche modo alla lista dei processi del sistema operativo (Task Manager su sistemi Windows) ed interrompere quello relativo all'applicazione Java.

Q.2.1 Struttura della libreria AWT ed esempi

La libreria **AWT** (acronimo di **Abstract Windows Toolkit**, che potremmo tradurre in italiano come **kit di strumenti astratto per creare finestre**) offre comunque una serie molto ampia di classi ed interfacce per la creazione di GUI. È possibile utilizzare pulsanti, checkbox, liste, combo box (classe `Choice`), label, radio button (utilizzando checkbox raggruppati mediante la classe `CheckboxGroup`), aree e campi di testo, scrollbar, finestre di dialogo (classe `Dialog`), finestre per navigare sul file system (classe `FileDialog`), etc. Per esempio, il seguente codice crea un'area di testo con testo iniziale `Java AWT`, 4 righe, 10 colonne e con la caratteristica di andare a capo automaticamente. La co-

stante statica `SCROLLBARS_VERTICAL_ONLY` infatti verrà interpretata dal costruttore in modo tale da utilizzare solo scrollbar verticali e non orizzontali in caso di sforamento.

```
TextArea ta =
  new TextArea("Java AWT", 4, 10, TextArea.SCROLLBARS_VERTICAL_ONLY);
```

Il numero di colonne è puramente indicativo. Per un certo tipo di font, una "w" potrebbe occupare lo spazio di tre "i".

La libreria AWT, dipendendo strettamente dal sistema operativo, definisce solamente i componenti grafici che appartengono all'intersezione comune dei sistemi operativi più diffusi. Per esempio, l'albero (in inglese "tree") di Windows (vedi "esplora risorse"), non esistendo su tutti i sistemi operativi, non è contenuto in AWT.
È molto semplice creare menu personalizzati tramite le classi `MenuBar`, `Menu`, `MenuItem`, `CheckboxMenuItem` ed eventualmente `MenuShortcut` per utilizzarli direttamente con la tastiera mediante le cosiddette scorciatoie. Segue un semplice frammento di codice che crea un piccolo menu:

```
Frame f = new Frame("MenuBar");
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
MenuItem mi1 = new MenuItem("New");
MenuItem mi2 = new MenuItem("Open");
MenuItem mi3 = new MenuItem("Save");
MenuItem mi4 = new MenuItem("Quit");
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);
m1.addSeparator();
m1.add(mi4);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
```

Non ci dovrebbe essere bisogno di spiegazioni.

Si noti come il menu Help sia stato aggiunto, diversamente dagli altri, mediante il metodo `setHelpMenu()`. Questo perché su un ambiente grafico come il CDE di Solaris, il menu di Help viene piazzato all'estrema destra della barra dei menu e su altri sistemi potrebbe avere posizionamenti differenti. Per quanto semplice sia l'argomento, quindi, è comunque necessario dare sempre uno sguardo alla documentazione prima di utilizzare una nuova classe.

La classe `Toolkit` permette di accedere a varie caratteristiche, grafiche e non grafiche, del sistema su cui ci troviamo. Per esempio, il metodo `getScreenSize()` restituisce un oggetto `Dimension` con all'interno la dimensione dello schermo. Inoltre offre il supporto per la stampa tramite il metodo `getPrintJob()`. Per ottenere un oggetto `Toolkit` possiamo utilizzare il metodo `getDefaultToolkit()`:

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
```

AWT offre anche la possibilità di utilizzare i font di sistema o personalizzati, tramite la classe `Font`. Per esempio, quando deve stampare un file, EJE utilizza il seguente `Font`:

```
Font font = new Font("Monospaced", Font.BOLD, 14);
```

Con AWT è anche possibile disegnare. Infatti ogni `Component` può essere esteso e si può sottoporre a override il metodo `paint(Graphics g)` che ha la responsabilità di disegnare il componente stesso. In particolare, la classe `Canvas` (in italiano "tela") è stata creata proprio per diventare un componente da estendere allo scopo di disegnarci sopra. Come esempio quindi creeremo proprio un oggetto `Canvas`:

```
public class MyCanvas extends Canvas {
    public void paint(Graphics g) {
        g.drawString("java",10,10);
        g.setColor(Color.red);
        g.drawLine(10,5, 35,5);
    }
}
```

La classe precedente stampa la scritta `java` con una linea rossa che l'attraversa. Se poi guardiamo la documentazione di `Graphics` troveremo diversi metodi per disegnare ovali, rettangoli, poligoni etc. Con la classe `Color` si possono anche creare colori ad hoc con lo standard RGB (red, green e blue), specificando l'intensità di ciascun colore con valori compresi tra 0 e 255:

```
Color c = new Color (255, 10 ,110 );
```

In figura Q.2 vi è una parte significativa della gerarchia di classi di AWT.

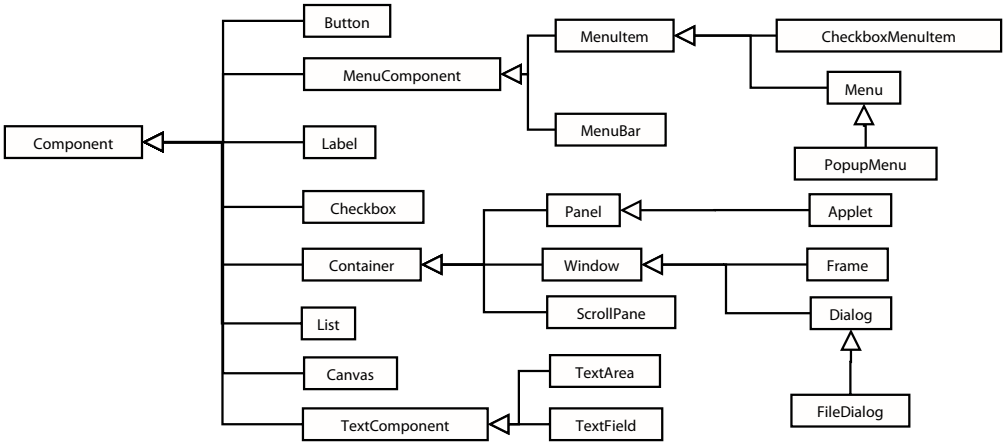


Figura Q.2 - Gerarchia di classi di AWT (Composite Pattern).

Si tratta di un'implementazione del pattern strutturale GoF chiamato **Composite**. La classe astratta `Component` astrae il concetto di componente generico astratto. Quindi, ogni componente grafico è sottoclasse di `Component` (ed è quindi un component). `Button`, e `Checkbox` sono sottoclassi dirette di `Component` e ridefiniscono il metodo `paint()`. Come già asserito, questo ha il compito di disegnare il componente stesso e quindi viene implementato nuovamente in tutte le sottoclassi di `Component`. Tra le sottoclassi di `Component`, bisogna però notarne una un po' particolare: la classe `Container`. Questa astrae il concetto di componente grafico astratto che può contenere altri componenti. Non è una classe astratta, ma solitamente vengono utilizzate le sue sottoclassi `Frame` e `Panel`.

Le applicazioni grafiche si basano sempre su un "top level container", ovvero un container di primo livello. In ogni applicazione Java con interfaccia grafica è necessario quantomeno istanziare un top level container, di solito un `Frame`, anche se fosse nascosto. Un caso speciale è rappresentato dalle applet, di cui parleremo nel paragrafo Q.5.

La caratteristica chiave dei container è avere un metodo `add(Component c)` che consente di aggiungere altri componenti come `Button`, `Checkbox` ma anche altri container (che essendo sottoclasse di `Component` sono anch'essi `Component`). Per esempio, è possibile aggiungere `Panel` a `Frame`. Il primo problema che si pone è: dove posiziona il componente aggiunto il container, se ciò non viene specificato esplicitamente? La risposta è nella prossima unità didattica.

Q.3 Creazione di interfacce complesse con i layout manager

La posizione di un componente aggiunto a un container dipende essenzialmente dall'oggetto che è associato al container, detto *layout manager*. In ogni container, infatti, esiste un layout manager associato per default. Un layout manager è un'istanza di una classe che implementa l'interfaccia `LayoutManager`. Esistono decine di implementazioni di `LayoutManager`, ma le più importanti sono solo cinque:

- `FlowLayout`;
- `BorderLayout`;
- `GridLayout`;
- `CardLayout`;
- `GridBagLayout`.

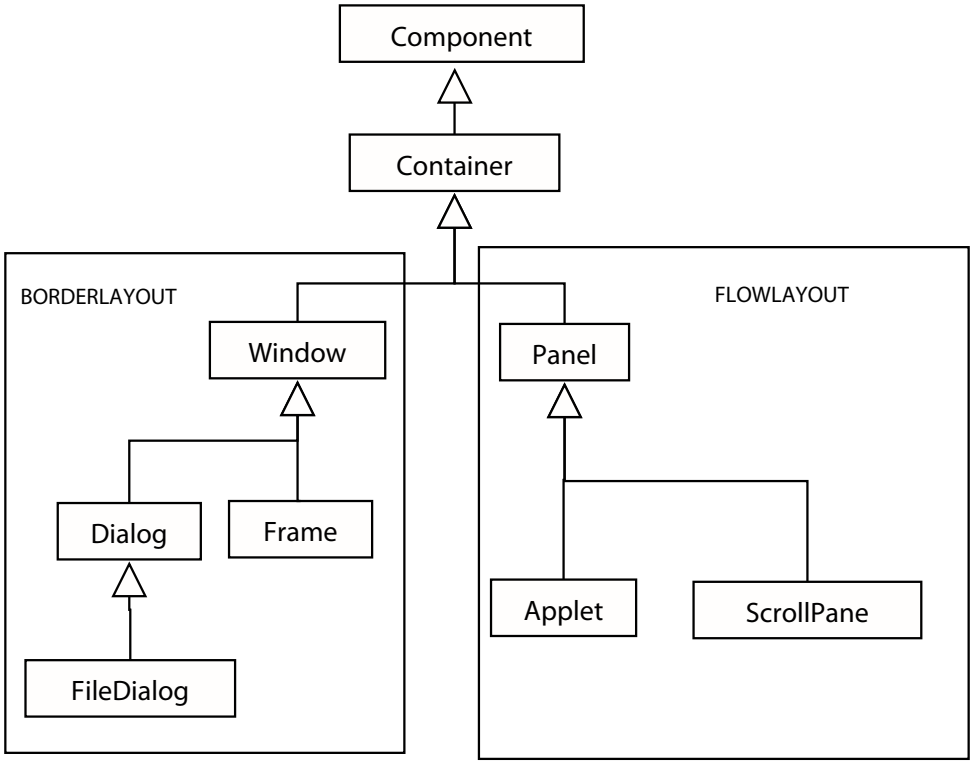


Figura Q.3 - Layout manager associati di default.

In questa appendice introdurremo le prime quattro, accennando solo al `GridBagLayout`. Come al solito il lettore interessato potrà approfondire lo studio di quest'ultima classe con la documentazione Oracle.

Anche la dimensione dei componenti aggiunti dipenderà dal layout manager.

La figura Q.3 mostra come tutte le sottoclassi di `Window` (quindi anche `Frame`) abbiano associato per default il `BorderLayout`, mentre tutta la gerarchia di `Panel` utilizza il `FlowLayout` per il posizionamento dei componenti.

È anche possibile non utilizzare layout manager per gestire interfacce grafiche. Tale tecnica però comprometterebbe la consistenza e la portabilità della GUI stessa. Il lettore interessato può provare per esempio ad annullare il layout di un container (per esempio un `Frame`) con l'istruzione `setLayout(null)` e poi usare i metodi `setLocation()`, `setBounds()` e `setSize()` per gestire il posizionamento dei componenti.

Q.3.1 Il `FlowLayout`

Il `FlowLayout` è il layout manager di default di `Panel`, e come vedremo è una delle classi principali del package AWT. `FlowLayout` dispone i componenti aggiunti in un flusso ordinato che va da sinistra a destra con un allineamento centrato verso l'alto. Per esempio, il codice seguente (da inserire all'interno di un metodo `main()`):

```
Frame f = new Frame("FlowLayout");
Panel p = new Panel();
Button button1 = new Button("Java");
Button button2 = new Button("Windows");
Button button3 = new Button("Motif");
p.add(button1);
p.add(button2);
p.add(button3);
f.add(p);
f.pack();
f.setVisible(true);
```

produrrebbe come output quanto mostrato in figura Q.4. Si noti che il metodo `pack()` semplicemente ridimensiona il frame in modo tale da mostrarsi abbastanza grande da visualizzare il suo contenuto.

Figura Q.4 - Il FlowLayout in azione.

In particolare, le figure Q.5 e Q.6 mostrano anche come si dispongono i pulsanti dopo avere ridimensionato la finestra che contiene il `Panel`. In figura Q.5 è possibile vedere come l'allargamento della finestra non alteri la posizione dei pulsanti sul `Panel`.

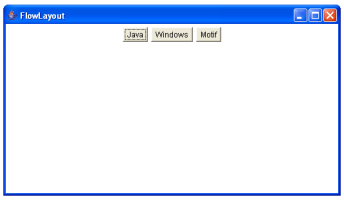


Figura Q.5 - Il FlowLayout dopo aver allargato il frame.

Mentre in figura Q.6 è possibile vedere come i pulsanti si posizionino in maniera coerente con la filosofia del `FlowLayout`, in posizioni diverse dopo aver ristretto molto il frame.

Figura Q.6 - Il FlowLayout dopo aver ristretto il frame.

Il `FlowLayout` utilizza per i componenti aggiunti la loro dimensione preferita. Infatti, tutti i componenti ereditano dalla classe `Component` il metodo `getPreferredSize()` (in italiano "dammi la dimensione preferita"). Il `FlowLayout` chiama questo metodo per ridimensionare i componenti prima di aggiungerli al container. Per esempio, il metodo `getPreferredSize()` della classe `Button` dipende dall'etichetta che gli viene impostata. Un pulsante con etichetta OK avrà dimensioni molto più piccole rispetto ad un pulsante con etichetta Ciao io sono un bottone AWT.

Q.3.2 Il `BorderLayout`

Il `BorderLayout` è il layout manager di default per i `Frame`, il top level container per eccellenza. I componenti sono disposti solamente in cinque posizioni specifiche che si ridimensionano automaticamente:

- `NORTH`, `SOUTH` che si ridimensionano orizzontalmente;
- `EAST`, `WEST` che si ridimensionano verticalmente;
- `CENTER` che si ridimensiona orizzontalmente e verticalmente.

Questo significa che un componente aggiunto in una certa area si deformerà per occupare l'intera area. Segue un esempio:

```
import java.awt.*;

public class BorderExample {
    private Frame f;
    private Button b[] = { new Button("b1"), new Button("b2"),
                           new Button("b3"), new Button("b4"),
                           new Button("b5")
                         };
    public BorderExample() {
        f = new Frame("Border Layout Example");
    }
    public void setup() {
        f.add(b[0], BorderLayout.NORTH);
        f.add(b[1], BorderLayout.SOUTH);
        f.add(b[2], BorderLayout.WEST);
        f.add(b[3], BorderLayout.EAST);
        f.add(b[4], BorderLayout.CENTER);
```

```
            f.setSize(200, 200);
            f.setVisible(true);
        }
        public static void main(String args[]) {
            new BorderExample().setup();
        }
    }
```

La figura Q.7 mostra l'output della precedente applicazione.

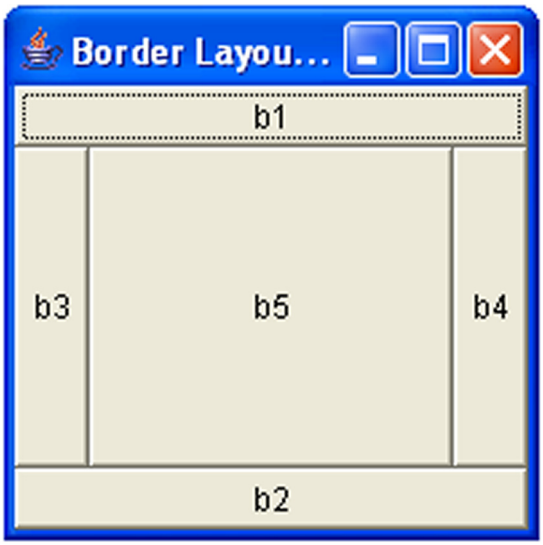


Figura Q.7 - Il BorderLayout in azione.

Quando si aggiungono componenti con il `BorderLayout` quindi, si utilizza il metodo `add(Component c, int position)` oppure `add(Component c, String position)`. Se si utilizza il metodo `add(Component c)` il componente sarà aggiunto al centro dal `BorderLayout`.

Q.3.3 Il `GridLayout`

Il `GridLayout` dispone i componenti da sinistra verso destra e dall'"alto verso il basso all'interno di una griglia. Tutte le regioni della griglia hanno sempre la stessa dimensione (anche se vengono modificate le dimensioni del frame) e i componenti occuperanno tutto lo spazio possibile all'interno delle varie regioni. Il costruttore del `GridLayout` permette di specificare righe e colonne della griglia. Come per il `BorderLayout` i componenti occuperanno interamente le celle in cui vengono aggiunti. Il seguente codice mostra come può essere utilizzato il `GridLayout`:

```
import java.awt.*;

public class GridExample {
    private Frame f;
    private Button b[] = { new Button("b1"), new Button("b2"),
                           new Button("b3"), new Button("b4"),
                           new Button("b5"), new Button("b6")
                         };
    public GridExample() {
        f = new Frame("Grid Layout Example");
    }
    public void setup() {
        f.setLayout(new GridLayout(3, 2));
        for (int i=0; i<6; ++i) {
                f.add(b[i]);
        }
        f.setSize(200, 200);
        f.setVisible(true);
    }
    public static void main(String args[]) {
        new GridExample().setup();
    }
}
```

La figura Q.8 mostra l'output della precedente applicazione.

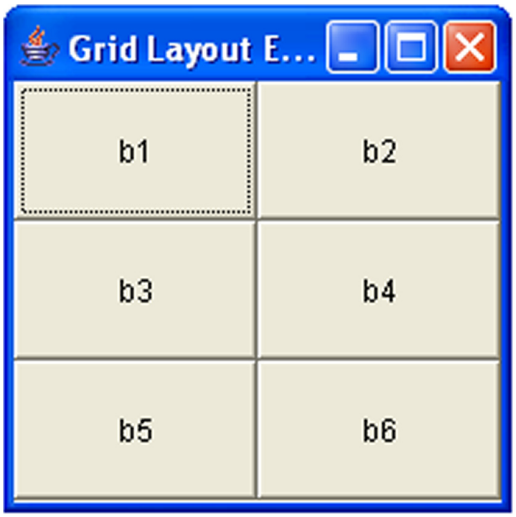


Figura Q.8 - Il GridLayout in azione.

Per quanto riguarda il costruttore di `GridLayout` che abbiamo appena utilizzato nell'esempio, è possibile specificare che il numero delle righe o delle colonne può essere zero (ma non può essere zero per entrambi). Lo zero viene inteso come "un numero qualsiasi di oggetti che può essere piazzato in una riga o una colonna".

Q.3.4 Creazione di interfacce grafiche complesse

Cerchiamo ora di capire come creare GUI con layout più complessi. È possibile infatti sfruttare i vari layout in un'unica GUI, creando così un layout composito, complesso e stratificato. In un `Frame`, per esempio, possiamo inserire molti container (come i `Panel`), che a loro volta possono disporre i componenti mediante il proprio layout manager. Il seguente codice mostra come creare una semplice interfaccia per uno strano editor:

```
import java.awt.*;

public class CompositionExample {
    private Frame f;
    private TextArea ta;
    private Panel p;
    private Button b[] = { new Button("Open"), new Button("Save"),
                           new Button("Load"), new Button("Exit")
                         };
    public CompositionExample() {
        f = new Frame("Composition Layout Example");
        p = new Panel();
        ta = new TextArea();
    }
    public void setup() {
        for (int i=0; i<4; ++i) {
            p.add(b[i]);
        }
        f.add(p, BorderLayout.NORTH);
        f.add(ta, BorderLayout.CENTER);
        f.setSize(350, 200);
        f.setVisible(true);
    }
    public static void main(String args[]) {
        new CompositionExample().setup();
    }
}
```

La figura Q.9 mostra l'output della precedente applicazione. Componendo i layout tramite questa tecnica, è possibile creare un qualsiasi tipo di interfaccia.

Il consiglio in questo caso è progettare con schizzi su un foglio di carta tutti gli strati che dovranno comporre l'interfaccia grafica. È difficile creare una GUI senza utilizzare questa tecnica, che tra l'altro suggerisce anche eventuali container riutilizzabili.

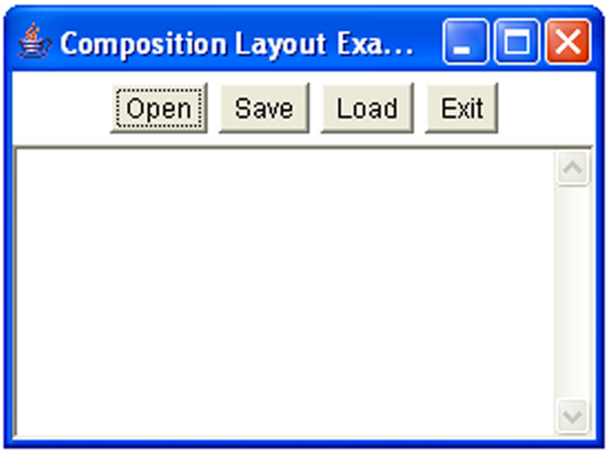


Figura Q.9 - L'interfaccia per un semplice editor.

Q.3.5 Il `GridBagLayout`

Il `GridBagLayout` può organizzare interfacce grafiche complesse da solo. Infatti, anch'esso è capace di dividere il container in una griglia ma, a differenza del `GridLayout`, può disporre i suoi componenti in modo tale che si estendano anche oltre un'unica cella. Quindi, anche nella più complicata delle interfacce, è idealmente possibile dividere in tante celle il container quanti sono i pixel dello schermo e piazzare i componenti a proprio piacimento. Anche se quella appena descritta non è una soluzione praticabile, rende l'idea della potenza del `GridBagLayout`. Si può tranquillamente affermare che da solo il `GridBagLayout` può sostituire i tre precedenti layout manager di cui abbiamo parlato. In compenso però la difficoltà di utilizzo è notevole. Per tale ragione non descriveremo in questa sede i dettagli di questa classe, rimandando il lettore interessato alla lettura della documentazione ufficiale.

Q.3.6 Il `CardLayout`

Il `CardLayout` è un layout manager particolare che permetterà di posizionare i vari componenti uno sopra l'altro, come le carte in un mazzo. Il seguente esempio mostra come è possibile disporre i componenti utilizzando un `CardLayout`:

```
import java.awt.*;

public class CardTest {
    private Panel p1, p2, p3;
    private Label lb1, lb2, lb3;
    private CardLayout cardLayout;
    private Frame f;
```

```
    public CardTest() {
        f = new Frame ("CardLayout");
        cardLayout = new CardLayout();
        p1 = new Panel();
        p2 = new Panel();
        p3 = new Panel();
        lb1 = new Label("Primo pannello rosso");
        p1.setBackground(Color.red);
        lb2 = new Label("Secondo pannello verde");
        p2.setBackground(Color.green);
        lb3 = new Label("Terzo pannello blue");
        p3.setBackground(Color.blue);
    }
    public void setup() {
        f.setLayout(cardLayout);
        p1.add(lb1);
        p2.add(lb2);
        p3.add(lb3);
        f.add(p1, "uno");
        f.add(p2, "due");
        f.add(p3, "tre");
        cardLayout.show(f, "uno");
        f.setSize(200, 200);
        f.setVisible(true);
    }
    private void slideShow() {
        while (true) {
            try {
                Thread.sleep(3000);
                cardLayout.next(f);
            }
            catch (InterruptedException exc) {
                exc.printStackTrace();
            }
        }
    }
    public static void main (String args[]) {
        CardTest cardTest = new CardTest();
        cardTest.setup();
        cardTest.slideShow();
    }
}
```

Tre pannelli sono aggiunti sfruttando un `CardLayout` e ad ogni panel viene assegnato un alias ("`uno`", "`due`" e "`tre`"). Viene impostato il primo pannello da visualizzare con l'istruzione:

```
cardLayout.show(f, "uno");
```

e dopo aver visualizzato la GUI viene invocato il metodo `slideShow()` che, trami-

te il metodo `next()`, mostra con intervalli di tre secondi i vari panel. Per far questo viene utilizzato il metodo `sleep()` della class `Thread`, già incontrato nel capitolo 15 dedicato ai thread.

Figura Q.10 - Il card layout che alterna i tre pannelli.

Solitamente l'alternanza di pannelli realizzata nel precedente esempio non viene governata da un thread in maniera temporale. Piuttosto sembra sempre più evidente che occorre un modo per interagire con le GUI. Questo *modo di interagire* è descritto nella prossima unità didattica.

Q.4 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. Nel capitolo 10, quando sono state introdotte le classi innestate e le classi anonime, si è data anche una descrizione della storia di come si è arrivati alla definizione della gestione degli eventi in Java. Si parla di *modello a delega* e si tratta di un'implementazione nativa del pattern GoF noto come **Observer**.

Q.4.1 Classi innestate: introduzione e storia

Quando nel 1995 la versione 1.0 di Java fu introdotta nel mondo della programmazione si parlava di linguaggio orientato agli oggetti *puro*. Ciò non era esatto. Un linguaggio orientato agli oggetti puro, come SmallTalk, non dispone di tipi di dati primitivi, ma solo di classi da cui istanziare oggetti. Dunque non esistono operatori nella sintassi. Proviamo ad immaginare cosa significhi utilizzare un linguaggio che per sommare due numeri interi costringe ad istanziare due oggetti dalla classe `Integer` ed invocare il metodo `sum()` della classe `Math`, passandogli come parametri i due oggetti. È facile intuire perché SmallTalk non abbia avuto lo stesso successo di Java. Java invece voleva essere un linguaggio orientato agli oggetti, ma anche semplice da apprendere. Di conseguenza non ha eliminato i tipi di dati primitivi e gli operatori. Ciononostante, il supporto che il nostro linguaggio offre

ai paradigmi della programmazione ad oggetti, come abbiamo avuto modo di apprezzare, è notevole. Ricordiamo che l'Object Orientation è nata come scienza che vuole imitare il mondo reale, giacché i programmi rappresentano un tentativo di simulare concetti fisici e matematici importati dalla realtà che ci circonda. C'è però da evidenziare un aspetto importante delle moderne applicazioni object oriented. Come già accennato in precedenza, solitamente dovremmo dividere un'applicazione in tre parti distinte:

- una parte rappresentata da ciò che è visibile all'utente, chiamata **view**, ovvero l'interfaccia grafica (in inglese "GUI": Graphical User Interface);
- una parte che rappresenta i dati e le funzionalità dell'applicazione, ovvero il **model**;
- una parte che gestisce la logica di controllo dell'applicazione, detto **controller**.

Partizionare un'applicazione come descritto implica notevoli vantaggi per il programmatore. Per esempio nel debug semplifica la ricerca dell'errore. Quanto appena riportato non è altro che una banalizzazione di uno dei più importanti Design Pattern conosciuti, noto come pattern **Model–View–Controller** o, più brevemente, **MVC** Pattern (una descrizione del pattern in questione la si può trovare all'indirizzo http://www.claudiodesio.com/download/mvc.zip, dove è possibile anche scaricare un esempio con codice sorgente). L'MVC propone questa soluzione architetturale per motivi molto profondi ed interessanti, e la soluzione è molto meno banale di quanto si possa pensare. L'applicazione di questo modello implica che l'utente utilizzi l'applicazione, generando eventi (come il clic del mouse su un pulsante) sulla view, che saranno gestiti dal controller per l'accesso ai dati del model. Il lettore può immaginare come in questo modello le classi che costituiscono il model, la view ed il controller abbiano ruoli ben definiti. L'Object Orientation supporta l'intero processo d'implementazione dell'MVC, ma c'è un'eccezione: la view. Infatti, se un'applicazione rappresenta un'astrazione idealizzata della realtà, l'interfaccia grafica non costituisce imitazione della realtà. La GUI esiste solamente nel contesto dell'applicazione stessa, "all'interno del monitor".
In tutto questo discorso si inseriscono le ragioni della nascita delle classi innestate e delle classi anonime nel linguaggio. Nella versione 1.0 infatti, Java definiva un modello per la gestione degli eventi delle interfacce grafiche noto come **modello gerarchico** (in inglese **hierarchical model**). Esso effettivamente non distingueva in maniera netta i ruoli delle classi costituenti la view ed il controller di un'applicazione. Di conseguenza avevamo la possibilità di scrivere classi che astraevano

componenti grafici, i quali avevano anche la responsabilità di gestire gli eventi da essi generati. Per esempio, un pulsante di un'interfaccia grafica poteva contenere il codice che doveva gestire gli eventi di cui era sorgente (bastava riscrivere il metodo `action()` ereditato dalla superclasse `Component`). Una situazione del genere non rendeva certo giustizia alle regole dell'astrazione. Ma la verità è che non esiste un'astrazione reale di un concetto che risiede all'interno delle applicazioni!
In quel periodo, se da una parte erano schierati con Sun Microsystems per Java, grandi società come Netscape e IBM, dall'altra parte era schierata Microsoft. Un linguaggio indipendente dalla piattaforma non era gradito al monopolista dei sistemi operativi. In quel periodo provennero attacchi da più fonti verso Java. Si mise in dubbio addirittura che Java fosse un linguaggio object oriented! Lo sforzo di Sun si concentrò allora nel risolvere ogni ambiguità, riscrivendo una nuova libreria di classi (e di interfacce). Nella versione 1.1 fu definito un nuovo modello di gestione degli eventi, noto come **modello a delega** (in inglese **delegation model**). Questo nuovo modo per gestire gli eventi permette di rispettare ogni regola dell'object orientation. Il problema nuovo però, riguarda la complessità di implementazione del nuovo modello, che implica la scrittura di codice tutto sommato superfluo. Infatti, un'applicazione che deve gestire eventi con la filosofia della delega richiede la visibilità da parte di più classi sui componenti grafici della GUI. Ecco che allora, contemporaneamente alla nascita del modello a delegazione, fu definita anche una nuova struttura dati: la classe innestata ("nested class"). Il vantaggio principale delle classi innestate risiede proprio nel fatto che esse hanno visibilità *agevolata* sui membri della classe dove sono definite. Quindi, per quanto riguarda le interfacce grafiche, è possibile creare un gestore degli eventi di una certa GUI come classe innestata all'interno della classe che rappresenta la GUI stessa. In questo modo sarà risparmiato tutto il codice di incapsulamento delle variabili d'istanza della GUI (di solito sono pulsanti, pannelli, etc.) che (solo in casi del genere) è superfluo. Non è sempre utile incapsulare un pulsante. Inoltre verrà risparmiato da parte dei gestori degli eventi il codice di accesso a tali variabili d'istanza, che in molti casi potrebbe essere abbastanza noioso.

Q.4.2 Observer e Listener

Anche se il pattern si chiama Observer (osservatore) in quest'appendice parleremo soprattutto di **Listener** (ascoltatore). Il concetto è lo stesso e sembra che il nome sia diverso perché, quando è stato creato il nuovo modello a delega nella versione 1.1 di Java, già esisteva una classe `Observer` (che serviva proprio per implementare "a mano" il pattern). Con il modello a delega esistono almeno tre oggetti per gestire gli eventi su una GUI:

1. il componente sorgente dell'evento (in inglese "event source");
2. l'evento stesso;
3. il gestore dell'evento, detto **listener**.

Per esempio se premiamo un pulsante e vogliamo che appaia una scritta su una `Label` della stessa interfaccia, allora:

1. il pulsante è la sorgente dell'evento;
2. l'evento è la pressione del pulsante, che sarà un oggetto istanziato direttamente dalla JVM dalla classe `ActionEvent`;
3. il gestore dell'evento sarà un oggetto istanziato da una classe a parte che implementa un'interfaccia `ActionListener` (in italiano "ascoltatore d'azioni"). Quest'ultima ridefinirà il metodo `actionPerformed(ActionEvent ev)` con il quale sarà gestito l'evento. Infatti, la JVM invocherà automaticamente questo metodo su quest'oggetto quando l'utente premerà il pulsante.

Sarà anche necessario effettuare un'operazione supplementare: *registrare* il pulsante con il suo ascoltatore. È necessario infatti istruire la JVM su quale oggetto invocare il metodo di gestione dell'evento. Ma vediamo in dettaglio come questo sia possibile:

```
import java.awt.*;

public class DelegationModel {
    private Frame f;
    private Button b;

    public DelegationModel() {
        f = new Frame("Delegation Model");
        b = new Button("Press Me");
    }

    public void setup() {
        b.addActionListener(new ButtonHandler());
        f.add(b, BorderLayout.CENTER);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String args[]) {
        DelegationModel delegationModel = new DelegationModel();
        delegationModel.setup();
    }
}
```

L'unica istruzione che ha bisogno di essere commentata è:

```
b.addActionListener(new ButtonHandler());
```

Trattasi della *registrazione* tra il pulsante ed il suo gestore. Dopo tale istruzione la JVM sa su quale oggetto di tipo `Listener` chiamare il metodo `actionPerformed()`. Il metodo `addActionListener()` si aspetta come parametro un oggetto di tipo `ActionListener`, ed essendo quest'ultima un'interfaccia, significa che `addActionListener()` si aspetta un oggetto istanziato da una classe che implementa tale interfaccia. La classe di cui stiamo parlando e che gestisce l'evento (il listener) è la seguente:

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

public class ButtonHandler implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        System.out.println("È stato premuto il bottone");
        System.out.println("E la sua etichetta è: "
          + e.getActionCommand());
    }
}
```

Ogni volta che viene premuto il pulsante, verrà stampata sulla riga di comando l'etichetta ("`Press Me!`") del pulsante stesso, mediante il metodo `getActionCommand()`. Vista così non sembra una grossa impresa gestire gli eventi. Basta:

1. creare la GUI;
2. creare un listener;
3. registrare il componente interessato con il rispettivo listener.

Al resto pensa la JVM. Infatti, alla pressione del pulsante viene istanziato un oggetto di tipo `ActionEvent` (che viene riempito di informazioni riguardanti l'evento) e passato in input al metodo `actionPerformed()` dell'oggetto listener associato; un meccanismo che ricorda da vicino quello già studiato delle eccezioni. In quel caso, l'evento era l'eccezione (anch'essa veniva riempita di informazioni su ciò che era avvenuto) ed al posto del metodo `actionPerformed()` c'era un blocco `catch`.
Nell'esempio appena visto c'è una enorme e vistosa semplificazione. La scritta, piuttosto che essere stampata sulla stessa GUI, viene stampata sulla riga di comando. Qualcosa di veramente originale creare un'interfaccia grafica per stampare sui prompt dei comandi...
Ma cosa dobbiamo fare se vogliamo stampare la frase in una label della stessa GUI? Come può procurarsi i *reference giusti* la classe `ButtonHandler`? Proviamo a

fare un esempio. Stampiamo la frase su un oggetto `Label` della stessa GUI:

```
import java.awt.*;

public class TrueDelegationModel {

    private Frame f;
    private Button b;
    private Label l;

    public TrueDelegationModel() {
        f = new Frame("Delegation Model");
        b = new Button("Press Me");
        l = new Label();
    }

    public void setup() {
        b.addActionListener(new TrueButtonHandler(l));
        f.add(b, BorderLayout.CENTER);
        f.add(l, BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }

    public static void main(String args[]) {
        TrueDelegationModel delegationModel = new TrueDelegationModel();
        delegationModel.setup();
    }
}
```

Notiamo come ci sia un cambiamento notevole: quando è istanziato l'oggetto listener `TrueButtonHandler`, viene passata al costruttore la label.

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

public class TrueButtonHandler implements ActionListener {

    private Label l;
    private int counter;

    public TrueButtonHandler(Label l) {
        this.l = l;
    }

    public void actionPerformed(ActionEvent e) {
        l.setText(e.getActionCommand() + " - " + (++counter));
    }
}
```

Come è facile osservare, il codice si è notevolmente complicato. La variabile `counter` è stata utilizzata per rendere evidente l'evento di pressione sul pulsante.

Questo tipo di approccio può scoraggiare lo sviluppatore. Troppo codice per fare qualcosa di semplice, e tutta colpa dell'incapsulamento! Ma come affermato precedentemente in questo testo, quando si programmano le GUI, si possono prendere profonde licenze rispetto all'Object Orientation. Non è un caso infatti che le classi innestate siano nate insieme al modello a delega (versione 1.1 di Java) e le classi anonime ancora più tardi (versione 1.2).

Q.4.3 Classi innestate e classi anonime

Nel capitolo 10 sono stati introdotti due argomenti, le classi innestate e le classi anonime, ed in quest'appendice probabilmente ne apprezzeremo ancor più l'utilità. Ricordiamo brevemente che una classe innestata è definita come una classe definita all'interno di un'altra classe. Per quanto riguarda la gestione degli eventi, l'implementazione del gestore dell'evento tramite una classe innestata rappresenta una soluzione molto vantaggiosa. Segue il codice dell'esempio precedente rivisitato con una classe innestata:

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

public class InnerDelegationModel {

    private Frame f;
    private Button b;
    private Label l;

    public InnerDelegationModel() {
        f = new Frame("Delegation Model");
        b = new Button("Press Me");
        l = new Label();
    }

    public void setup() {
        b.addActionListener(new InnerButtonHandler());
        f.add(b, BorderLayout.CENTER);
        f.add(l, BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
```

```
    }

    public class InnerButtonHandler implements ActionListener {
        private int counter;
        public void actionPerformed(ActionEvent e) {
            l.setText(e.getActionCommand() + " - " + (++counter));
        }
    }

    public static void main(String args[]) {
        InnerDelegationModel delegationModel = new InnerDelegationModel();
        delegationModel.setup();
    }
}
```

Si può notare come la proprietà delle classi innestate di vedere le variabili della classe esterna come se fosse pubblica abbia semplificato il codice. Infatti, nella classe `InnerButtonHandler` non è più presente il reference alla `Label l`, né il costruttore che serviva per impostarla, visto che è disponibile direttamente il reference "originale".
Una soluzione ancora più potente è rappresentata dall'utilizzo di una classe anonima per implementare il gestore dell'evento:

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

public class AnonymousDelegationModel {

    private Frame f;
    private Button b;
    private Label l;
    private int counter;

    public AnonymousDelegationModel() {
        f = new Frame("Delegation Model");
        b = new Button("Press Me");
        l = new Label();
    }

    public void setup() {
        b.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                l.setText(e.getActionCommand() + " - " +(++counter));
            }
        } );
        f.add(b, BorderLayout.CENTER);
        f.add(l, BorderLayout.SOUTH);
        f.pack();
```

```
        f.setVisible(true);
    }

    public static void main(String args[]) {
        AnomynousDelegationModel delegationModel =
          new AnomynousDelegationModel();
        delegationModel.setup();
    }
}
```

La classe anonima presenta una sintassi sicuramente più compatta, che esula dai soliti standard (cfr. capitolo 10), ma quando ci si abitua è difficile rinunciarvi. Inoltre, rispetto ad una classe innestata, una classe anonima è sempre un singleton. Infatti la sua sintassi obbliga ad istanziare una ed una sola istanza, il che è una soluzione ottimale per un gestore degli eventi. Una buona programmazione ad oggetti richiederebbe che ogni evento abbia il suo *gestore personale.*

Una limitazione delle classi anonime è non poter avere un costruttore (il costruttore ha lo stesso nome della classe). È possibile però utilizzare un inizializzatore d'istanza (cfr. capitolo 10) per inizializzare una classe anonima, al quale però non si possono passare parametri. Per un esempio d'utilizzo reale di un inizializzatore d'istanza, si possono studiare le classi anonime del file `EJE.java` **(file sorgenti di EJE scaricabili all'indirizzo** `http://sourceforge.net/projects/eje` **nella sezione download), in particolare le classi che rappresentano le azioni ed estendono la classe** `AbstractAction` **(che poi implementa** `ActionListener`**, cfr. documentazione di** `AbstractAction`**).**

Q.4.4 Espressioni lambda

Con Java 8 e l'avvento delle espressioni lambda, anche le classi anonime potrebbero diventare semplicemente storia. La sintassi compatta delle espressioni lambda semplifica ancora di più l'implementazione dei gestori degli eventi. Ecco l'esempio precedente che fa uso di un'espressione lambda al posto della classe anonima.

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

public class LambdaDelegationModel {
```

```
    private Frame f;
    private Button b;
    private Label l;
    private int counter = 0;

    public LambdaDelegationModel() {
        f = new Frame("Delegation Model");
        b = new Button("Press Me");
        l = new Label();
    }

    public void setup() {
        b.addActionListener(
          e -> l.setText(e.getActionCommand() + " - " +(++counter))
        );
        f.add(b, BorderLayout.CENTER);
        f.add(l, BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }

    public static void main(String args[]) {
        LambdaDelegationModel delegationModel =
          new LambdaDelegationModel();
        delegationModel.setup();
    }
}
```

Consultare il capitolo 16 per le espressioni lambda.

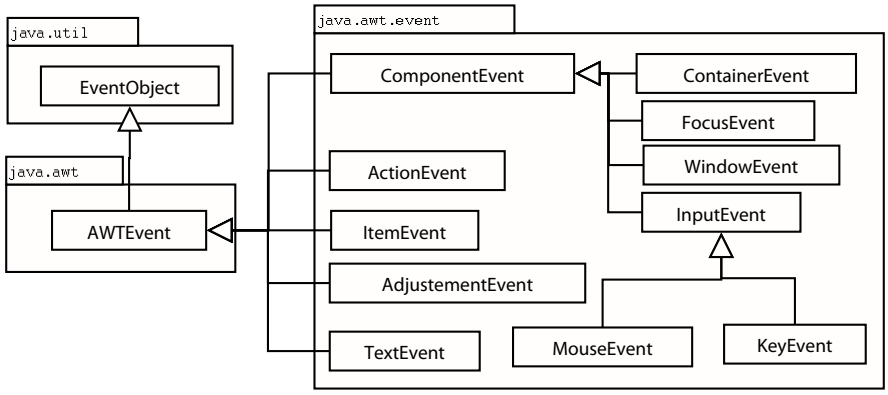


Figura Q.11 - La gerarchia delle classi evento.

Q.4.5 Altri tipi di eventi

Come già accennato precedentemente, esistono vari tipi di eventi che possono essere generati dai componenti e dai container di una GUI. Per esempio si potrebbe gestire l'evento di una pressione di un tasto della tastiera, il rilascio del pulsante del mouse, l'acquisizione del focus da parte di un certo componente e soprattutto la chiusura della finestra principale. Esiste nella libreria una gerarchia di classi di tipo evento che viene riportata sommariamente in figura Q.11.
Nella seguente tabella, invece, vengono riportate una schematizzazione del tipo di evento, una veloce descrizione, l'interfaccia per la sua gestione e i metodi che sono dichiarati da essa. Ci limiteremo solo agli eventi più importanti.

Evento	Descrizione	Interfaccia	Metodi
`ActionEvent`	Azione (generica)	`ActionListener`	`actionPerformed`
`ItemEvent`	Selezione	`ItemListener`	`itemStateChanged`

`MouseEvent`	Azioni effettuate con il mouse	`MouseListener`	`mousePressed`
			`mouseReleased`
			`mouseEntered`
			`mouseExited`
			`mouseClicked`

`MouseEvent`	Movimenti del mouse	`MouseMotionListener`	`mouseDragged` `mouseMoved`

`KeyEvent`	Pressione di tasti	`KeyListener`	`keyPressed`
			`keyReleased`
			`keyTyped`

`WindowEvent`	Azioni effettuate su finestre	`WindowListener`	`windowClosing`
			`windowOpened`
			`windowIconified`
			`windowDeiconified`
			`windowClosed`
			`windowActivated`
			`windowDeactivated`

Non tutti i componenti possono generare tutte le tipologie di eventi. Per esempio, un pulsante non può generare un `WindowEvent`.
Come primo esempio, vediamo finalmente come chiudere un'applicazione grafica semplicemente chiudendo il frame principale. Supponendo che `frame` sia il reference del frame principale, il seguente frammento di codice implementa una

classe anonima che definendo il metodo `windowClosing()` (cfr. documentazione) consente all'applicazione di terminarsi:

```
frame.addWindowListener( new WindowListener() {

    public void windowClosing(WindowEvent ev) {
        System.exit(0);
    }

    public void windowClosed(WindowEvent ev) {}

    public void windowOpened(WindowEvent ev) {}

    public void windowActivated(WindowEvent ev) {}

    public void windowDeactivated(WindowEvent ev) {}

    public void windowIconified(WindowEvent ev) {}

    public void windowDeiconified(WindowEvent ev) {}
} );
```

Purtroppo, implementando l'interfaccia `WindowListener`, abbiamo ereditato ben sette metodi dovendo riscriverli tutti, pur utilizzandone solo uno!

Q.4.6 Classi Adapter

Il problema appena visto nel paragrafo precedente viene mitigato, almeno in parte, dall'esistenza delle classi dette **adapter**. Un adapter è una classe che implementa un listener, riscrivendo ogni metodo ereditato senza codice applicativo. Per esempio, la classe `WindowAdapter` è implementata più o meno come segue:

```
public abstract class WindowAdapter implements WindowListener {

    public void windowClosing(WindowEvent ev) {}

    public void windowClosed(WindowEvent ev) {}

    public void windowOpened(WindowEvent ev) {}

    public void windowActivated(WindowEvent ev) {}

    public void windowDeactivated(WindowEvent ev) {}

    public void windowIconified(WindowEvent ev) {}

    public void windowDeiconified(WindowEvent ev) {}
}
```

Quindi, se al posto di implementare l'interfaccia listener si estende la classe adapter, non abbiamo più bisogno di riscrivere tutti i metodi ereditati, ma solo quelli che ci interessano. La nostra classe anonima diventerà molto più compatta:

```
frame.addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent ev) {
        System.exit(0);
    }
} );
```

L'estensione di un adapter rispetto alla implementazione di un listener è un vantaggio solo per il numero di righe da scrivere. Infatti il nostro gestore di eventi eredita comunque i metodi vuoti, il che non rappresenta una soluzione object oriented molto corretta.

Non sempre è possibile sostituire l'estensione di un adapter all'implementazione di un listener. Un adapter è comunque una classe e ciò impedirebbe l'estensione di eventuali altre classi. Come vedremo nella prossima unità didattica, per esempio, per un'applet non è possibile utilizzare un adapter, dato che per definizione si deve già estendere la classe `Applet`. Ricordiamo che è invece possibile implementare più interfacce e quindi qualsiasi numero di listener. Per esempio, se dovessimo creare una classe che gestisse sia la chiusura della finestra che la pressione di un certo pulsante, si potrebbe utilizzare una soluzione mista adapter-listener come la seguente:

```
public class MixHandler extends WindowAdapter implements ActionListener {
    public void actionPerformed(ActionEvent e) {...}
    public void windowClosing(WindowEvent e) {...}
}
```

Con la nuova definizione di interfaccia di Java 8, dove è possibile definire metodi di default, sarebbe stato possibile anche eliminare del tutto il bisogno delle classi adapter. Tuttavia, per la compatibilità all'indietro, si è preferito lasciare tutto com'era.

Q.4.7 Errori classici

Di seguito qualche utile raccomandazione per evitare errori che abbiamo visto commettere molte volte.

1. `Listener` si scrive con le `e` tra la `t` e la `n`. Purtroppo, spesso noi italiani scriviamo i termini inglesi non molto noti, così come li pronunciamo.
2. Se avete scritto del codice ma l'evento non viene minimamente gestito, come primo tentativo di correzione controllate se avete registrato il componente con il relativo gestore. Spesso infatti, presi dalla logica della gestione, ci si dimentica del passaggio della registrazione.
3. `WindowEvent`, `WindowListener` e tutti i suoi metodi (`windowClosing()`, `windowActivated()`, etc.) si riferiscono al concetto di finestra (in inglese "window") e non al sistema operativo Microsoft! Quindi attenzione a non cadere nell'abitudine di scrivere `windowsClosing()` al posto di `windowClosing()`, perché vi potrebbe portar via molto tempo in debug. Se per esempio state utilizzando un `WindowAdapter` e sbagliate l'override di uno dei suoi metodi come appena descritto, il metodo che avete scritto semplicemente non sarà mai chiamato. Verrà invece chiamato il metodo vuoto della superclasse adapter che però non farà niente, neanche avvertirvi del problema.

Q.5 La classe `Applet`

L'applet API è stata **deprecata** in Java 9. I browser non supporteranno in futuro (ed alcuni già non supportano più) il Java plugin per eseguire applet. Oracle consiglia di migrare ad altre tecnologie come applicazioni desktop da scaricare e Java Web Start.

Java Web Start è una tecnologia che permette di scaricare un'applicazione Java, sempre aggiornata, facendo clic su un link di una pagina web. Per informazioni consultare la seguente pagina: https://docs.oracle.com/javase/9/deploy/java-web-start-technology.htm.

A tal proposito è possibile consultare i seguenti indirizzi:

- https://docs.oracle.com/javase/9/deploy/migrating-java-applets-jnlp.htm: per migrare alla tecnologia Java Web Start;
- https://docs.oracle.com/javase/9/deploy/self-contained-application-packaging.htm: per migrare ad applicazioni desktop da scaricare ed eseguire direttamente da file system.

Nonostante l'Applet API stia scomparendo definitivamente, e vista comunque l'importanza storica della tecnologia, abbiamo preferito non eliminare questo paragrafo. Chi non è interessato, può tranquillamente saltare al successivo paragrafo. Si tratta di una tecnologia che ha dato grande impulso e pubblicità a Java nei primi tempi, ancora oggi molto utilizzata sulle pagine web (cfr. appendice A).

In inglese, il termine "applet" potrebbe essere tradotta come applicazioncina". Questo perché una applet deve essere un'applicazione leggera dovendo essere scaricata dal Web insieme alle pagine HTML. Pensiamo anche al fatto che nel 1995, anno di nascita di Java ma anche delle applet, non esistevano connessioni a banda larga. Oltre alla dimensione, un'applet deve anche subire il caricamento, i controlli di sicurezza e l'interpretazione della JVM. Quindi è bene che sia piccola. In queste pagine si parlerà delle applet al femminile. Altri autori preferiscono parlare di applet al maschile.

Le applet sono eseguite direttamente da pagine web; ovvero un'applet è un'applicazione Java che può essere direttamente connessa ad una pagina HTML mediante un tag speciale: il tag `<APPLET>`. Introdurremo brevemente quindi anche il linguaggio HTML.

Q.5.1 Definizione di Applet

Un'**applet**, per definizione, deve estendere la classe `Applet` del package `java.applet`. Ne eredita i metodi e può sottoporli a override. L'applet non ha infatti un metodo `main()` bensì i metodi ereditati sono invocati direttamente dalla JVM del browser seguendo determinate regole. Quindi, se il programmatore riscrive i metodi opportunamente, riuscirà a far eseguire all'applet il codice che vuole. Segue una prima banale applet contenente commenti esplicativi:

```
import java.applet.*;
import java.awt.*;
public class BasicApplet extends Applet {
    public void init() {
    // Chiamato una sola volta dal browser appena viene
    // eseguita l'applet
    }
    public void start() {
    // Chiamato ogni volta che la pagina che contiene
    // l'applet diviene visibile
    }
    public void paint(Graphics g) {
    // Chiamato ogni volta che la pagina che contiene
    // l'applet deve essere disegnata
    }
    public void stop() {
    // Chiamato ogni volta che la pagina che contiene
    // l'applet diviene non visibile
    }
    public void destroy() {
    // Chiamato una sola volta dal browser quando viene
    // distrutta l'applet
    }
}
```

Tenendo conto di quanto scritto nei commenti, il programmatore deve gestire l'esecuzione dell'applet. Non è obbligatorio scrivere tutti i cinque metodi, ma almeno uno si dovrebbe sottoporre a override. Per esempio, la seguente applet stampa una parola:

```
import java.applet.*;
import java.awt.*;
public class StringApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Applet", 10, 10);
    }
}
```

L'oggetto `Graphics`, come accennato precedentemente, mette a disposizione molti metodi per il disegno. Per poter eseguire un'applet, però, bisogna anche creare una pagina HTML che la inglobi. Introduciamo quindi brevemente cos'è il linguagggio HTML.

Q.5.2 Introduzione allo HTML

HTML è l'acronimo per **HyperText Markup Language**, ovvero linguaggio di marcatura per ipertesti. Un **ipertesto** è una tipologia di documento che, oltre ad includere semplice testo, possiede collegamenti ad altre pagine. Non si tratta di un

vero linguaggio di programmazione, ma solo di un linguaggio di formattazione di pagine. Non esistono per esempio strutture di controllo come `if` o `for`. Le istruzioni dello HTML si dicono **tag**. I tag (in italiano "etichette") sono semplici istruzioni con la seguente sintassi:

```
<NOME_TAG [LISTA DI ATTRIBUTI]>
```

Dove ogni attributo, opzionale, ha una sintassi del tipo:

```
CHIAVE=VALORE
```

Il valore di un attributo potrebbe e dovrebbe essere compreso tra due apici o due virgolette. Ciò è però necessario se e solo se il valore è costituito da più parole separate. Inoltre, quasi tutti i tag HTML, salvo alcune eccezioni, vanno chiusi con una istruzione del tipo:

```
</NOME_TAG>
```

Ad esempio, il tag:

```
<HTML>
```

va chiuso con:

```
</HTML>
```

Mentre il tag:

```
<applet code='StringApplet' width='100' height='100'>
```

va chiuso con:

```
</applet>
```

Una semplice pagina HTML si può scrivere all'interno di un file testuale con suffisso .htm o .html. Non occorre altro che un semplice editor di testo come Blocco Note di Windows. Un browser come Mozilla Firefox, Google Chrome e così via, può poi farci visualizzare la pagina formattata. Salviamo quindi il file HelloWorld.html che contiene i seguenti tag:

```
<HTML>
  <HEAD>
    <TITLE>Test HTML</TITLE>
  </HEAD>
  <BODY bgcolor='green'>
    <CENTER>
      <H1>
        Hello
```

```
          <BR/>
          HTML World!
        </H1>
      </CENTER>
    </BODY>
  </HTML>
```

In questo esempio sono dichiarati i tag HTML, HEAD, TITLE, BODY e CENTER. Il nome del tag è sempre rinchiuso tra parentesi acute, ed un tag è opzionalmente composto da un tag di apertura e un tag di chiusura. Quest'ultimo è riconoscibile dal simbolo / che precede il nome del tag. L'unico tag che non è dichiarato con un'apertura e una chiusura è il tag BR, che è dichiarato con un simbolo / che questa volta segue il nome del tag, e che vuole indicare che il tag rappresenta un tag di apertura e chiusura. Quando l'apriremo all'interno di un browser allora vedremo una pagina web dal titolo "Test HTML", con la scritta "Hello World!" centrata in alto, su uno sfondo di colore verde (vedi figura Q.12). Il browser infatti sa interpretare i tag HTML e formatta la pagina di conseguenza.

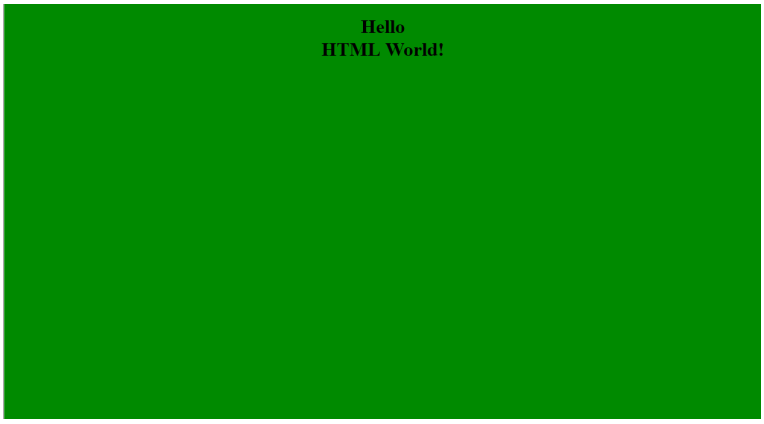


Figura Q.12 - Pagina HTML aperta all'interno del browser Mozilla Firefox.

Infatti i tag dell'esempio hanno un significato che i browser sanno interpretare. In particolare il tag HTML indica che si sta dichiarando un file HTML, e deve contenere tutto il codice HTML. Il tag HEAD (che in italiano possiamo tradurre come "testata") contiene altri tag che rappresentano informazioni sul documento. Nel nostro caso contiene il tag TITLE, che contiene il titolo del documento. Chiuso il tag HEAD, si apre il tag BODY, che contiene il corpo visibile del documento. Notiamo che BODY dichiara anche un **attributo** chiamato BGCOLOR. (che sta per "background color", ovvero "colore dello sfondo"). All'interno del tag di apertura infatti, viene dichiarato una coppia *chiave=valore* che specifica che il colore di sfondo deve essere verde

(green). Ciò che è contenuto nel tag `CENTER` avrà un allineamento centrato rispetto alla pagina. Il testo contenuto nel tag `H1` avrà una dimensione di tipo `H1` (esistono anche i tag `H2`, `H3`, `H4`, `H5`, `H6`) e rappresenta la dimensione massima del testo. Infine il tag `BR`, che come abbiamo notato non è costituito da un tag di apertura e chiusura, rappresenta solo il comando per andare da capo (se fossimo semplicemente andati a capo il browser non avrebbe mandato a capo la scritta "HTML WORLD").

HTML non è case-sensitive.

Non resta altro che procurarci un manuale HTML (da Internet se ne possono scaricare centinaia), o semplicemente imparare da pagine già fatte andando a spiarne il codice che è sempre disponibile tramite browser.

Q.5.3 Distribuire l'applet

Ritornando alla nostra applet, basterà creare un semplice file con suffisso .htm o .html, che contenga il seguente testo:

```
<applet code='StringApplet' width='100' height='100'>
</applet>
```

Al variare degli attributi `width` ed `height`, varierà la dimensione dell'area che la pagina HTML dedicherà all'applet. È anche possibile passare alla pagina HTML dei parametri che verranno letti all'esecuzione dall'applet sfruttando il meccanismo esplicitato nel seguente esempio. Aggiungendo il seguente override del metodo `init()` all'esempio precedente, è possibile parametrizzare per esempio la frase da stampare:

```
import java.applet.*;
import java.awt.*;
public class ParameterApplet extends Applet {
    String s;
    @Override
    public void init() {
        String parameterName = "p";
        s = getParameter(parameterName);
    }
    @Override
    public void paint(Graphics g) {
        g.drawString(s, 10, 10);
    }
}
```

Il codice della pagina HTML che deve caricare l'applet precedente cambierà

leggermente:

```
<applet code='ParameterApplet' width='100' height='100'>
  <param name='p' value='Java'/>
</applet>
```

Per approfondire l'argomento rimandiamo il lettore interessato al Java Tutorial di Oracle (cfr. bibliografia) o alle migliaia di esempi disponibili in rete.

Dalla versione 6 in poi Java offre supporto ai linguaggi di scripting (Javascript, Python, Ruby, etc.). Anche se l'argomento non è propriamente standard (bensì riguarderebbe di più un discorso Java Enterprise) è comunque da segnalare. Con Java è possibile mixare per esempio codice Javascript e codice Java, cosa molto utile per creare prototipi, o in ambienti dove coesistono persone con skill eterogenei. Con Java 8 il vecchio motore Javascript Rhino, è stato sostituito con il ben più performante e completo Nashorn.

Q.6 Introduzione a Swing

Swing è il nome della libreria grafica di seconda generazione di Java.

Se il lettore cercherà una panoramica sui componenti più importanti di Swing in questo paragrafo, non la troverà. Infatti, per quanto si possa essere precisi nella descrizione di un componente Swing, la documentazione rappresenterà comunque la guida migliore per lo sviluppatore. La complessità di questa libreria dovrebbe sempre obbligare il programmatore ad utilizzare la documentazione. Per esempio, alcuni componenti di Swing come le tabelle o le liste (classi `JTable` e `JList`) utilizzano una sorta di pattern MVC per separare i dati dalla logica di accesso ad essi.

Swing fa parte di un gruppo di librerie note come **Java Foundation Classes** (**JFC**). Le JFC oltre a Swing, includono:

1. **Java 2D**: una libreria che permette agli sviluppatori di incorporare grafici di alta qualità, testo, effetti speciali ed immagini all'interno di applet ed applicazioni.
2. **Accessibility**: una libreria che consente a strumenti diversi dai soliti monitor (per esempio schermi Braille) di accedere alle informazioni sulle GUI.
3. **Supporto al Drag and Drop** (DnD): una serie di classi che permettono di gestire il trascinamento dei componenti grafici.
4. **Supporto al Pluggable Look and Feel**: offre la possibilità di cambiare lo stile delle GUI che utilizzano Swing, e più avanti ne vedremo un esempio.

I componenti AWT sono stati forniti già dalla prima versione di Java (JDK 1.0), mentre Swing è stata inglobata come libreria ufficiale solo dalla versione 1.2 in poi. Si raccomanda fortemente l'utilizzo di Swing piuttosto che di AWT nelle applicazioni Java. Swing ha infatti molti *pro* ed un solo *contro* rispetto ad AWT. Essendo quest'ultimo abbastanza importante, introdurremo questa libreria partendo proprio da questo argomento.

Q.6.1 Swing vs AWT

A differenza di AWT, Swing non effettua chiamate native al sistema operativo dirette per sfruttarne i componenti grafici, bensì li ricostruisce da zero. Questa caratteristica di Swing rende AWT nettamente più performante. Swing è di sicuro la causa principale per cui Java gode della fama di essere un linguaggio lento. Esempi di applicazioni che utilizzano questa libreria sono proprio i più famosi strumenti di sviluppo come NetBeans. Per esempio, eseguendo Netbeans, passeranno diversi secondi (o addirittura minuti se non si dispone di una macchina con risorse sufficienti) per poter finalmente vedere la GUI. Una volta caricata, però, l'applicazione gira ad una velocità più che accettabile (dando per scontato di avere una macchina al passo con i tempi). Questo perché il problema principale risiede proprio nel caricamento dei componenti della GUI e bisogna tenere conto che una GUI complessa potrebbe essere costituita da centinaia di componenti. Una volta caricati, le azioni su di essi non richiedono lo stesso sforzo da parte del sistema. Per esempio, se l'apertura di un menu scritto in C++ (linguaggio considerato altamente performante) richiede un tempo nell'ordine di millesimi di secondo, la stessa azione effettuata su un equivalente menu scritto in Java (a parità di macchina) potrebbe al massimo essere eseguita nell'ordine del decimo di secondo. Questa differenza è poco percepibile dall'occhio umano. Rimane comunque il problema di dover aspettare un po' di tempo prima di vedere le GUI Swing.

Come già asserito nell'appendice A, il problema può essere risolto solamente con il tempo, ed i miglioramenti tanto degli hardware che della Virtual Machine. Ma per chi ha visto (come noi) Java in azione già nel 1995 con gli hardware di allora, la situazione attuale è più che soddisfacente. D'altronde, la programmazione dei nostri giorni (anche con altri linguaggi compresi quelli di .Net e C++) tende a badare meno alle performance e sempre più alla robustezza del software... in questo senso Java ha precorso i tempi.

Attualmente le performance della Java Virtual Machine e degli hardware moderni, hanno reso i tempi di attesa irrilevanti rispetto a programmi scritti in altri linguaggi.

Esiste anche una terza libreria grafica non ufficiale che fu sviluppata originariamente da IBM e poi donata al mondo open source: SWT. È una libreria sviluppata in C++ altamente performante e dallo stile piacevole, che è necessario inglobare nelle nostre applicazioni. Essendo scritta in C++, ne esistono versioni diverse per sistemi operativi diversi. Un esempio di utilizzo di SWT è l'interfaccia grafica di Eclipse (http://www.eclipse.org).

Anche EJE utilizza Swing ma, a differenza dei già citati IDE più popolari (Netbeans per esempio), ha una GUI estremamente più semplice e leggera, e del resto è solo uno strumento adatto all'apprendimento e non professionale.
Il fatto che Swing non utilizzi codice nativo porta anche diversi vantaggi rispetto ad AWT. Abbiamo già asserito come AWT possa definire solo un minimo comune denominatore dei componenti grafici presenti sui vari sistemi operativi. Questa non è più una limitazione per Swing. Swing definisce qualsiasi tipo di componente di qualsiasi sistema operativo, e addirittura ne inventa alcuni nuovi! Questo significa che sarà possibile per esempio vedere sul sistema operativo Solaris il componente **tree** (il famoso albero di "Esplora risorse") di Windows.
Inoltre ogni componente di Swing estende la classe `Container` di AWT e quindi può contenere altri componenti. Quindi ci sono tante limitazioni di AWT che

vengono superate. Per esempio:

- i pulsanti e le label di Swing possono visualizzare anche immagini oltre che semplice testo.
- Grazie al supporto del **pluggable look and feel** è possibile vedere GUI con stili di diversi sistemi operativi su uno stesso sistema operativo. Per esempio, EJE su di un sistema operativo Windows 7 permetterà di scegliere nelle opzioni (premere F12 e fare clic sul tab EJE) lo stile tra:
  - Metal (uno stile personalizzato di Java)
  - Windows
  - Windows Classic
  - CDE/Motif (stile familiare agli utenti Unix)
  - Nimbus, stile introdotto definitivamente con Java 7 basato sulla libreria Java2D

  Con AWT invece siamo costretti ad utilizzare lo stile della piattaforma nativa.
- È possibile facilmente cambiare l'aspetto o il comportamento di un componente Swing o invocando metodi o creando sottoclassi.
- I componenti Swing non devono per forza essere rettangolari. I `Button` possono per esempio essere circolari.
- Con il supporto della libreria `Accessibility` è semplice, per esempio, leggere con uno strumento come uno schermo Braille l'etichetta di una label o di un pulsante.
- È facile cambiare anche i bordi dei componenti con una serie di bordi predefiniti o personalizzati.
- È molto semplice anche utilizzare i **tooltip** (i messaggi descrittivi che appaiono quando si posiziona il puntatore su un componente) sui componenti Swing mediante il metodo `setToolTip()` e gestire il controllo delle azioni direttamente da tastiera.

Le classi di Swing, quindi, sono molto più complicate di quelle di AWT e consentono di creare interfacce grafiche senza nessun limite di fantasia.

È raccomandato di non mixare all'interno della stessa GUI componenti Swing con altri componenti "heavyweight" ("pesanti") di AWT. Per componenti pesanti si intendono tutti i componenti "pronti all'uso" come `Menu` e `ScrollPane` e tutti i componenti AWT che estendono le classi `Canvas` e `Panel`. Questa restrizione esiste perché, quando si sovrappongono i componenti Swing (e tutti gli altri componenti "lightweight") ai componenti pesanti, questi ultimi vengono sempre disegnati sopra. Con Java 7 questa limitazione è stata quasi eliminata del tutto, tranne che in alcuni casi.

I componenti Swing non sono *thread safe*. Infatti, se si modifica un componente Swing visibile, per esempio invocando su una label il metodo `setText()` da una qualsiasi parte di codice eccetto un gestore di eventi, allora bisogna prendere alcuni accorgimenti per rendere visibile la modifica. In realtà questo problema non si presenta spesso, perché nella maggior parte dei casi sono proprio i gestori degli eventi a implementare il codice per modificare i componenti.
Le classi di Swing si distinguono da quelle di AWT principalmente perché i loro nomi iniziano con una "`J`". Per esempio, la classe di Swing equivalente alla classe `Button` di AWT si chiama `JButton`. Per Swing il package di riferimento non è più `java.awt` ma `javax.swing`.

Si noti che, a differenza delle altre librerie finora incontrate (eccetto JAXP), il package principale non si chiama "`java`" ma "`javax`". La "`x`" finale sta per "eXtension" (gli americani decidono così le abbreviazioni e noi ci adeguiamo) perché inizialmente (JDK 1.0 e 1.1) Swing era solo un'estensione della libreria ufficiale.

Vi sono alcune situazioni in cui per ottenere un componente Swing equivalente a quello AWT non basterà aggiungere una `J` davanti al nome AWT. Per esempio, la classe equivalente a `Choice` di AWT si chiama `JComboBox` in Swing. Oppure, l'equivalente di `Checkbox` è `JCheckBox` con la "`B`" maiuscola.

Q.6.2 Le ultime novità per Swing

Detto che da Java 8 non si svilupperà più su Swing, le novità del framework risalgono a precedenti versioni. Dalla versione 6 infatti sono supportate alcune novità per quanto riguarda le interfacce grafiche. Mustang (nome in codice di Java 6) ha introdotto una facility per creare splash screen per le nostre applicazioni in maniera semplice e senza programmare. Prima della versione 6 il programmatore doveva lavorare sodo per poter creare uno splash screen. Ora è possibile utilizzare da riga di comando l'opzione "–splash" per il comando java. Basta indicare il percorso ad un'immagine e il gioco è fatto. Per esempio, con il seguente comando:

```
java –splash:miaImmagine.jpg mioProgrammaGrafico
```

soddisferemo l'impazienza dei nostri annoiati utenti.
Ma oltre agli splash screen Mustang ha introdotto tante altre novità nelle librerie di grafica. Per esempio è ora possibile accedere alle nostre applicazioni dal System Tray del nostro sistema operativo. Quindi sarà possibile portare nella parte in basso a destra del nostro sistema operativo la nostra applicazione, senza che sia per forza residente in una propria finestra. Il seguente esempio di codice mostra come sia possibile personalizzare il System Tray:

```
TrayIcon trayIcon = null;
if (SystemTray.isSupported()) {
    SystemTray tray = SystemTray.getSystemTray();
    Image image = Toolkit.getDefaultToolkit.getImage(...);
    ActionListener listener = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            //...
        }
    };
    PopupMenu popup = new PopupMenu();
    MenuItem defaultItem = new MenuItem(...);
    defaultItem.addActionListener(listener);
    popup.add(defaultItem);
    trayIcon = new TrayIcon(image, "Tray Demo", popup);
    trayIcon.addActionListener(listener);
    tray.add(trayIcon);}
//...
```

La documentazione della classe `SystemTray` (package `java.awt`) è sicuramente più esauriente. Inoltre sono state introdotte nel componente più complesso di Swing (`javax.swing.JTable`) altre funzionalità come l'ordinamento, il filtraggio dei dati e l'evidenziazione. A differenza di prima, ora i tab di un oggetto `javax.swing.JTabbedPane` possono essere `Component` e il drag and

drop di Swing è stato notevolmente migliorato. La classe astratta generica `javax.swing.SwingWorker` è stata aggiunta per consentire alle applicazioni Swing di eseguire task in background senza influire sulle funzionalità della GUI.
Java 7 oltre ad avere definitivamente introdotto il look and feel Nimbus ed aver eliminato quasi del tutto l'incompatibilità tra componenti heavyweight e lightweight, ha anche definito un nuovo componente denominato `JLayer` e la possibilità di usare la trasparenza delle finestre sui sistemi operativi che lo supportano. Per quanto riguarda `JLayer` si tratta di un nuovo componente che astrae il concetto di *strato* e che può decorare gli oggetti `Component`. Un `JLayer` può essere sovrapposto come componente invisibile alla nostra GUI e utilizzato per creare animazioni, disegni, effetti speciali oppure catturare eventi senza modificare la GUI stessa (cfr. documentazione di `JLayer`).

Q.6.3 File JAR eseguibile

I file JAR (Java Archive), come abbiamo visto nell'appendice E, sono spesso utilizzati per creare librerie da utilizzare nei programmi. C'è però un utilizzo meno noto dei file JAR ma molto di effetto: la creazione di un file JAR eseguibile. Per fare questo bisogna semplicemente scrivere nel file Manifest.mf una riga che permetta alla virtual machine di capire qual è la classe che definisce il metodo `main()`. Un esempio di contenuto di un file manifest potrebbe essere il seguente:

```
Manifest-Version: 1.0
Main-Class: com.claudiodesio.test.ClassConMain
```

Con l'ultima riga si istruisce la JVM in modo tale che possa avviare la nostra applicazione. Su un sistema Windows, quindi, un doppio clic sul file JAR avvierà l'applicazione come se fosse un normale eseguibile.

È possibile che abbiate installato sul vostro sistema Windows un programma come Winrar, associato all'apertura del file JAR. In tal caso dovete cambiare tale associazione, associando al comando javaw (compreso nella cartella bin del JDK) l'apertura del file JAR per poter sfruttare questa utilità.

Riepilogo

In quest'appendice abbiamo introdotto i principi per creare GUI al passo con i tempi ed in particolare abbiamo sottolineato l'importanza del pattern MVC. Abbiamo in seguito descritto la libreria AWT, sia elencando le sue caratteristiche principali, sia con una panoramica su alcuni dei suoi principali componenti. In particolare abbiamo sottolineato come tale libreria sia fondata sul pattern **Composite** e sui ruoli di **Component** e **Container**, senza però soffermarci troppo sui dettagli. La gestione del posizionamento dei componenti sul container è basata sul concetto di layout manager, di cui abbiamo introdotto i più importanti rappresentanti. La gestione degli eventi, invece, è basata sul modello a delega, a integrazione del quale abbiamo introdotto diversi concetti quali **adapter**, **classi innestate**, **anonime** ed **espressioni lambda**. Non poteva mancare anche un'introduzione alle applet, anche se l'intera libreria è stata deprecata e non viene supportata, o è supportata parzialmente, già da alcuni browser. Abbiamo anche introdotto brevemente il linguaggio HTML e soprattutto la libreria Swing, di cui abbiamo potuto apprezzarne la potenza.

Appendice R

Java Database Connectivity

Obiettivi:

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

- ✔ Saper spiegare la struttura dell'interfaccia JDBC (unità R.1, R.2).
- ✔ Saper scrivere codice che si connetta a qualsiasi tipo di database (unità R.2, R.3).
- ✔ Saper scrivere codice che aggiorni, interroghi e gestisca i risultati qualsiasi sia il database in uso (unità R.2, R.3).
- ✔ Saper spiegare le tipiche caratteristiche avanzate di JDBC, come stored procedure, statement parametrizzati e transazioni (unità R.3).
- ✔ Comprendere le differenze tra le varie versioni delle specifiche JDBC (unità R.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 le appendici H ed I) e sa interrogare ed aggiornare database tramite SQL. **SQL** (che è l'acronimo di **Structured Query Language** ovvero **linguaggio di interrogazione strutturato**) 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 ancora con un database relazionale mediante questo linguaggio. 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.

Per una miglior comprensione di quest'appendice, consigliamo al lettore di leggere anche l'appendice T, al fine di installare il database proposto oltre a testare gli esempi man mano che si incontrano.

R.1 Introduzione a JDBC

JDBC è 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, hanno oramai consolidato uno standard. JDBC, rispetto ad altre soluzioni, consente a 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** (**Database Management System**, ovvero **sistema di gestione del database**).

Caliamoci in uno scenario: supponiamo che una società abbia creato anni fa un'applicazione Java che utilizza un **RDBMS** (acronimo **Relational Database Management System**, ovvero **sistema di gestione del database relazionale**) come DB2, lo storico prodotto della IBM. La stessa applicazione viene eseguita su diverse piattaforme come Solaris, Windows e Linux. Ad un certo punto, per strategie aziendali, i responsabili decidono di sostituire DB2 con un altro RDBMS più economico, come PostgreSQL. A questo punto, l'unico sforzo da fare è migrare i dati da DB2 a PostgreSQL, mentre 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. Con Java e JDBC, le migrazioni ad altri database risultano invece molto meno costose.

Il nome "JDBC" deriva dall'altra tecnologia di connettività considerata standard: "ODBC". ODBC è l'acronimo di Open Database Connectivity ed è un tipo di connettività che tutti i database supportano di default.

R.2 Le basi di JDBC

Come già asserito, JDBC è uno strato di astrazione software tra un'applicazione Java ed un database. Questo significa che JDBC definisce le interfacce che utilizza l'applicazione (le interfacce non hanno metodi implementati), mentre ci sarà uno strato concreto che implementa tali interfacce per connettersi ad un particolare database. Quindi i nostri programmi non utilizzano direttamente le classi implementate, ma solo le interfacce definite da JDBC. Questo permette di cambiare database all'occorrenza, senza modificare l'applicazione. C'è da fare però una precisazione. La sua struttura a due livelli consente di accedere a RDBMS differenti, a patto che questi supportino l'**ANSI SQL 2** standard. Infatti, 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, tanto da poter essere distribuita con database differenti a clienti differenti, dovrebbe utilizzare solo comandi SQL standard, oppure prevedere appositi controlli laddove si vuole necessariamente adoperare un comando proprietario.
Un'applicazione Java che vuole utilizzare JDBC per interfacciarsi ad un database, è quindi composta da due componenti separati:

1. un'implementazione del fornitore del RDBMS (o di terze parti) conforme alle specifiche delle API `java.sql`. Tale implementazione viene solitamente chiamata **driver JDBC**, ed è costituita da una serie di classi che implementano le interfacce fondamentali di JDBC;
2. un'**implementazione** da parte dello sviluppatore dell'applicazione, che utilizza il driver JDBC senza dipendere da esso sfruttando l'astrazione fornita da JDBC.

R.2.1 Implementazione del fornitore (Driver JDBC)

Il fornitore deve fornire l'implementazione di una serie di interfacce definite dal package `java.sql` (appartenente al modulo `java.sql`, di cui in figura R.1 viene riportato il grafico delle dipendenze), 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:

```
nomeDBNomeInterfacciaImplementata
```

(per esempio: `DB2Driver`, `DB2Connection...`). Tale JAR viene comunemente detto **driver JDBC**. Il fornitore dovrebbe inoltre fornire allo sviluppatore anche una documentazione. Attualmente tutti i più importanti RDBMS supportano driver JDBC.

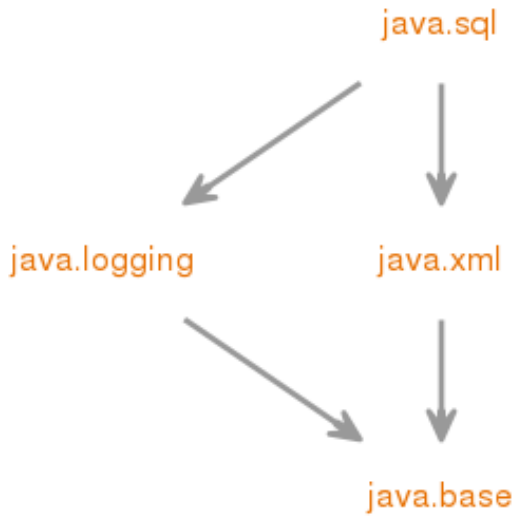


Figura R.1 - Grafico delle dipendenze del modulo java.sql.

Esistono quattro tipologie diverse di driver JDBC caratterizzati da differenti potenzialità e strutture.

- 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, e in generale questa tipologia di driver sta scomparendo.

- 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.
- 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.
- 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 eseguire gli esempi di quest'appendice, abbiamo scelto il database **Apache Derby**, un database flessibile, che era inglobato nelle precedenti versioni del JDK con il nome di Java DB.

Per le note di installazione e la configurazione di Apache Derby al fine di poter eseguire i programmi presentati in questa appendice, potete consultare l'appendice T. Intanto potete scaricare il software a questo indirizzo: https://db.apache.org/derby/derby_downloads.html.

R.2.2 Implementazione dello sviluppatore (Applicazione JDBC)

Lo sviluppatore ha un compito piuttosto semplice: implementare codice che sfrutta l'implementazione del fornitore, 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 il database Apache Derby:

```
 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              // Otteniamo 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          }
33      }
34  }
```

R.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 fornitore. In questo

modo l'applicazione non utilizzerà mai il nome di una classe concreta, rendendo in questo modo l'applicazione stessa indipendente dal database utilizzato. Infatti gli unici riferimenti espliciti all'implementazione del fornitore 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. capitolo 12) per caricare in memoria l'implementazione del driver JDBC, il cui nome completo è 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 di JDBC 4.1.

Tra la riga 9 e la riga 12 viene aperta una connessione al database mediante la definizione della stringa `url`, che deve essere disponibile nella documentazione fornita dal fornitore (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 si formatta una query SQL in una stringa chiamata `qry`, che viene poi eseguita sul database, e il cui risultato è immagazzinato 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 si effettua la chiusura di `ResultSet res`, `Statement cmd` e `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 fornitore del database direttamente dentro la `SQLException`. Tramite i codici di errori, i database come Oracle, riescono a specificare centinaia di errori diversi in maniera dettagliata. Una volta fornito il codice di errore infatti, è possibile consultare la documentazione del database, per trovare tutti i dettagli del problema.

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 altro tipo di database, bisognerà cambiare alcuni dettagli. In particolare sarà necessario:

- riga 6: assegnare alla stringa `driver` il nome del driver del database da utilizzare;
- riga 9: 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: 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)`.

R.2.4 Database schema

Lo schema del database usato dalla precedente applicazione, è stato chiamato `Music`, ed è stato creato con il seguente semplice script:

```
CREATE TABLE Album ( AlbumId int, Titolo varchar(20),
  Artista varchar(255), Anno int, PRIMARY KEY (AlbumId) );
INSERT INTO Album (AlbumId, Titolo, Artista, Anno)
  VALUES (1, 'Made In Japan','Deep Purple',1972);
INSERT INTO Album (AlbumId, Titolo, Artista, Anno)
  VALUES (2, 'Be','Pain Of Salvation',2004);
INSERT INTO Album (AlbumId, Titolo, Artista, Anno)
  VALUES (3, 'Images And Words','Dream Theater',1992);
INSERT INTO Album (AlbumId, Titolo, Artista, Anno)
  VALUES (4, 'The Human Equation','Ayreon',2004);
```

- Nell'appendice T viene spiegato come eseguire i comandi SQL. Una volta creato troverete i file generati all'interno della cartella utente, e dovrebbe avere un percorso simile a C:\users\NOME_UTENTE\Music.

R.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 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.

R.3.1 Indipendenza dal database

Avevamo asserito che la caratteristica più importante di un programma che utilizza 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 14.1.1) 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";
            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();
        }
    }
}
```

Dove il file di properties specificato config.properties contiene le seguenti proprietà:

```
jdbcUsername=username
jdbcPassword=password
jdbcDriver=org.apache.derby.jdbc.EmbeddedDriver
jdbcUrl=jdbc:derby:Music
```

Si noti che per cambiare database bisognerà modificare il driver e il file di configurazione, ma l'applicazione non cambierà di una virgola.

R.3.2 Operazioni CRUD

Con JDBC è possibile eseguire qualsiasi tipo di comando **CRUD** (acronimo di **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. Ciononostante 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 sono 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 dei 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) );
```

e 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 con la seguente istruzione:

```
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);
```

R.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.

R.3.4 Stored procedure

JDBC offre anche il supporto alle **stored procedure**, mediante la sotto-interfaccia `CallableStatement` di `PreparedStatement`. Segue un semplice 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();
}
```

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.

R.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 mostra 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
setASCIIStream	LONGVARCHAR prodotto da un ASCII stream
setBigDecimal	NUMERIC
setBinaryStream	LONGVARBINARY
setBoolean	BIT
setByte	TINYINT
setBytes	VARBINARY o LONGVARBINARY (dipende dai limiti relativi del VARBINARY)
setDate	DATE
setDouble	DOUBLE
setFloat	FLOAT
setInt	INTEGER
setLong	BIGINT
setNull	NULL
setObject	L'oggetto passato è convertito al tipo SQL corrispondente prima di essere impostato
setShort	SMALLINT
setString	VARCHAR o LONGVARCHAR (dipende dalla dimensione relativa ai limiti del driver sul VARCHAR)
setTime	TIME
setTimestamp	TIMESTAMP

R.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...
}
```

Nel caso fallisse una delle tre operazioni eseguite nel blocco `try` (supponiamo il `DELETE_STATEMENT`) sarà eseguito un rollback sulla transazione (chiamata al metodo `rollback()` dell'oggetto `Connection`) che annullerà anche le istruzioni di inserimento (`INSERT_STATEMENT`) e aggiornamento (`UPDATE_STATEMENT`) precedenti.

R.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 il JDK 9 ha introdotto la versione 4.3 dell'interfaccia JDBC. Questo non significa che dobbiamo utilizzare per forza tutte le novità di JDBC da subito necessariamente. Infatti la versione 4.3 include tutte le

altre versioni precedenti:

- ❑ JDBC 4.2
- ❑ 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 9	JDBC 4.3	JDK 9
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.

R.4.1 JDBC 2.0

Oramai praticamente tutti i fornitori 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 un "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 15.6.3.3.

In particolare i `DataSource` sono lo standard da utilizzare per le applicazioni lato server in ambienti Java EE (Enteprise Edition). Sostituiscono gli oggetti driver 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** (acronimo di **Java Naming and Directory Interface**, ovvero **interfaccia per nomi e directory di Java**) per ottenere un'istanza dell'oggetto `DataSource` allo scopo di sfruttare un'eventuale *connection pool* messo a disposizione direttamente da un server Java EE. In particolare JNDI tramite un'istanza di tipo `InitialContext`, ci permette di istanziare un'oggetto a partire da una stringa identificativa, tramite il metodo `lookup()`. Si può notare che viene specificata la stringa parametro `jdbc/myDataSource`, che rappresenta il nome (`myDataSource`) del `DataSource`, mentre `jdbc` rappresenta solo lo spazio di nomi dove JNDI deve cercare. Nel caso di `DataSource` sarà restituita un'istanza appartenente ad un pool di `DataSource`.

Questo sostituisce l'oggetto `Driver` nelle sue funzionalità, e ci permette di recuperare una `Connection` con il metodo `getConnection()`. Dopo avere ottenuto un oggetto `Connection`, i nostri passi per interagire con il database non cambiano.

R.4.2 JDBC 3.0

Oggigiorno 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 siano mostrati eventuali cambiamenti ai dati nel database (con cui il `ResulSet` 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", "Claudio");
rs.updateRow();
```

aggiorniamo nella terza riga del `ResultSet`, con il valore `Claudio`, 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, "Claudio");
rs.updateString("COGNOME", "De Sio Cesari");
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 database, ma che può anche sincronizzarsi con un'istruzione esplicita. Per esempio, con le seguenti istruzioni (dove `cachedRowSet` è un oggetto di tipo `CachedRowSet`):

```
cachedRowSet.updateString(2, "Claudio");
cachedRowSet.updateInt(4, 300);
cachedRowSet.updateRow();
cachedRowSet.acceptChanges();
```

il `CachedRowSet` si sincronizza con il database.

R.4.3 JDBC 4.x

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 dei **servizi con Service Provider** o anche **Java Standard Exension 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.

L'evoluzione di questo meccanismo in Java 9 lo abbiamo visto nel capitolo 19 con i moduli.

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`, l'unica riga del file META-INF/services/java.sql.Driver deve contenere la seguente istruzione:

```
my.sql.Driver
```

Questo è esattamente il caso del driver di Apache Derby:

```
org.apache.derby.jdbc.EmbeddedDriver
```

Ma non dobbiamo preoccuparci di implementare noi il meccanismo dell'estensione descritto in quanto è già pronto.

È possibile verificare quanto detto aprendo il file derby.jar che è posizionato nella cartella lib di Apache Derby.

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. Basterà utilizzare il metodo `getConnection()` di `DriverManager`.
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, grazie all'implementazione automatica di tutti i controlli che 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"/* ,
              "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`, 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.
Java 9 ha introdotto invece poche novità, di cui alcune troppo avanzate per poter essere spiegate in questa sede e quindi per ora trascurabili (per informazioni https://docs.oracle.com/javase/9/docs/api/java/sql/package-summary.html). Le novità più interessanti per chi legge quest'appendice sono l'introduzione di `DriverManager`, del metodo `drivers()` che restituisce uno `Stream` di oggetto `Driver` disponibili, e soprattutto l'introduzione dei metodi che aggiungono gli apici intorno ai parametri di tipo letterale nelle istruzioni SQL, come `enquoteLiteral()`.

Riepilogo

Quest'appendice è stata dedicata alla gestione dei database 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 questa appendice 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.

Appendice S

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 S.1).
- ✔ Saper elencare le caratteristiche di JavaFX (unità S.2).
- ✔ Saper scrivere semplici programmi con JavaFX (unità S.3).
- ✔ Saper gestire i principali Layout Pane per costruire GUI complesse (unità S.4).
- ✔ Essere in grado di creare semplici interfacce grafiche con il linguaggio FXML importandole in programmi JavaFX (unità S.4).
- ✔ Capire come associare file o istruzioni CSS in un programma JavaFX (unità S.5).
- ✔ Capire come creare gestori di eventi con il modello a delega di JavaFX (unità S.6).
- ✔ Saper definire proprietà JavaFX personalizzate e saper sfruttare la tecnica del binding (unità S.7).
- ✔ Imparare ad usare componenti grafici avanzati ed effetti speciali (unità S.8, S.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.

S.1 Storia delle GUI in Java

L'acronimo **GUI** sta per **Graphical User interface**, che in italiano si traduce **interfaccia grafica per l'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 esegui ovunque", che pubblicizzava la caratteristica di Java di essere un linguaggio multipiattaforma, fu la chiave del successo iniziale. Purtroppo le applet non mantennero la promessa. Infatti si basavano sulla libreria nota come AWT (acronimo di "Abstract Window Toolkit", cfr. appendice Q), la quale sfruttava i componenti grafici nativi del sistema operativo dove l'applet era eseguita. Dato che questi componenti differivano molto tra sistemi operativi diversi, la differenza tra interfacce grafiche tra sistema e sistema poteva risultare notevole. Quindi 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 programmatori non mostrarono 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 programmatori 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. Questo in quanto ogni piattaforma ha un modo molto differente di supportare la multimedialità rispetto alle altre, 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'era 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. Poi, con l'avvento di Java 8, anche JavaFX è stato direttamente promosso alla versione 8, e con Java 9 alla versione 9.
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 JavaFX 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 questa appendice introdurremo JavaFX per consentire 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! In quest'appendice 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 (a Swing e AWT è dedicata l'appendice Q). 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.

S.2 Caratteristiche di JavaFX

Java 9 ha riorganizzato la tecnologia JavaFX in 7 moduli: `javafx.base`, `javafx.controls`, `javafx.deploy`, `javafx.fxml`, `javafx.graphics`, `javafx.media`, `javafx.swing` e `javafx.web`.

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 risoluzioni oltre l'HD).
- Supporta il multi-touch sui dispositivi adeguati.
- Ingloba un motore di accelerazione grafica chiamato *Prism* che può essere utilizzato su diverse schede grafiche e GPU.

S.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 capitolo con Java: visualizziamo la scritta "Hello World!" con JavaFX. Il risultato è mostrato in figura S.1.

Figura S.1 - Hello World con JavaFX.

Il codice per eseguire il nostro programma "Hello World" è il seguente:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.text.Font;
import javafx.stage.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();
    }
}
```

S.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`). 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. Istanziamo 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 dal metodo `init()` che viene invocato prima del metodo `start()`, e dal metodo `stop()` invocato automaticamente quando l'applicazione viene chiusa.

Questo ricorda molto da vicino il ciclo di vita di un'applet.

È possibile verificare facilmente come funziona 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 utili e interessanti. 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).

S.3.2 Esecuzione di un'applicazione JavaFX

Le applicazioni JavaFX sono eseguite come ogni altra applicazione Java. 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. Potremmo creare un file batch (con suffisso .bat) scrivendo 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 batch. Altri modi alternativi sono: la creazione di un file JAR eseguibile e l'utilizzo di Java Web Start. Per i JAR eseguibili e le applet è possibile trovare dei paragrafi nell'appendice E, per Java Web Start è possibile consultare la pagina ufficiale di Oracle: https://docs.oracle.com/javase/9/deploy/java-web-start-technology.htm.

Come spiegato nell'appendice Q, le applet ormai non rappresentano più un'opzione percorribile.

S.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, mentre adesso creeremo interfacce ben più complesse. 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 un'etichetta, un campo di testo e un altro pannello nella parte inferiore che allinea un paio di pulsanti. In sostanza le GUI sono composte strato su strato, 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.

S.4.1 I Layout pane di JavaFX

Il discorso dei layout sarà familiare a chi ha già programmato interfacce grafiche con Java (cfr. appendice Q). 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`.

S.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 S.2.

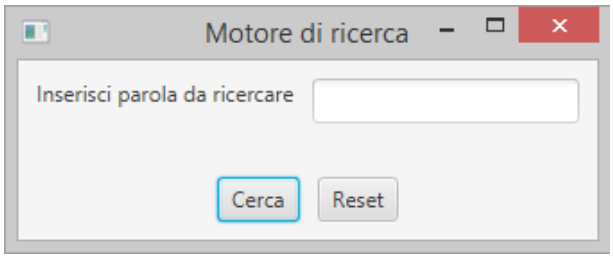


Figura S.2 - VBox e HBox.

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` li abbiamo aggiunti 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. 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 javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
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("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 javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

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();
    }
}
```

S.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 S.3.
Volendo potremmo inserire il pannello del motore di ricerca all'interno di un componente che estende il `BorderPane`:

```
import javafx.geometry.Pos;
```

```
import javafx.scene.control.Label;
import javafx.scene.layout.BorderPane;

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 S.4.

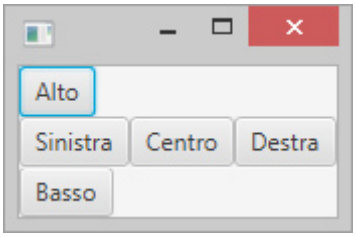


Figura S.3 - BorderPane.

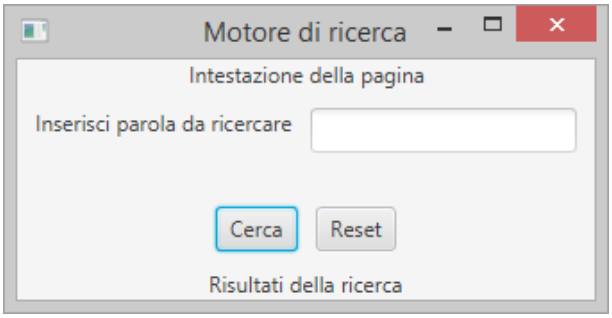


Figura S.4 - BorderPane + il componente del motore di ricerca.

S.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 sostanza è l'equivalente 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.setHalignment(cercaButton, HPos.RIGHT);
        GridPane.setHalignment(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 `setHalignment()` (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);
```

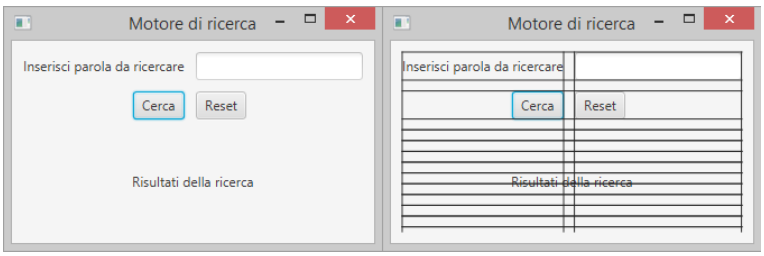


Figura S.5 - Motore di ricerca GUI con GridPane.

Dove abbiamo specificato che l'area dei risultati deve occupare dieci righe e due colonne. 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 S.5 è possibile osservare le due GUI affiancate, con e senza contorni.

S.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 su 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.

S.4.2 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://gluonhq.com/products/scene-builder. 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. Probabilmente non organizzato nella maniera che vorremmo, e senza i controlli necessari affinché l'interfaccia rimanga consistente al variare dei 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.

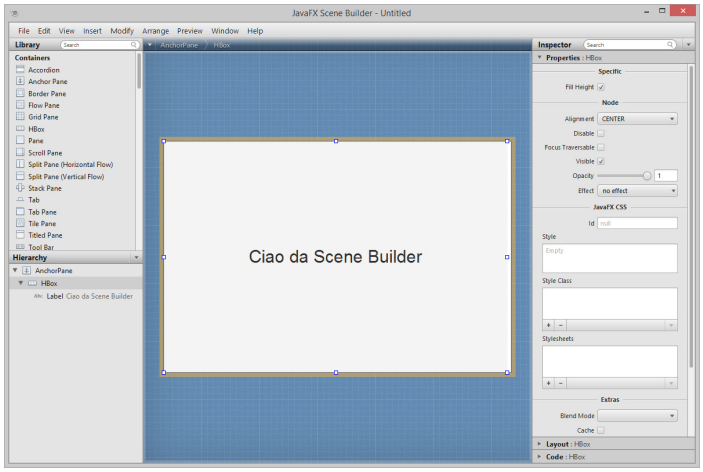


Figura S.6 - Scene Builder 2.0 in azione.

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 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`. In questo modo, con FXML, la logica che relaziona contenitore e contenuti risulta molto intuitiva. Il tag `children` e la relativa dichiarazione dei componenti grafici interni (molto riconoscibili 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 S.7.

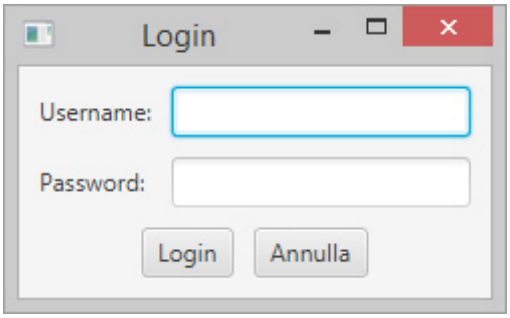


Figura S.7 - Login con FXML.

Riguardo il pattern **MVC** (spiegato brevemente nell'appendice Q dedicata a Swing/AWT, e che è possibile approfondire su un datato articolo che potete scaricare insieme al codice all'indirizzo http://www.claudiodesio.com/ooa&d/mvc.htm) Oracle dichiara che è possibile implementarlo usando FXML per codificare il componente **View**, e il codice Java per implementare il componente **Controller**.

S.5 CSS

Il **Cascading Style Sheets**, meglio noto come **CSS**, è un linguaggio che definisce la formattazione di linguaggi di markup come HTML e XML. Associare un file di stile ad un documento, significa poterne modificare la formattazione, senza modificare il contenuto testuale. Con JavaFX possiamo utilizzare tale linguaggio con le nostre interfacce. Aggiungere CSS ad un file FXML è molto semplice e naturale, infatti somiglia molto a quello che si fa con l'HTML.

S.5.1 CCS e file FXML

Per esempio potremmo creare un banale file CSS con una semplice istruzione di tipo classe, 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 degli equivalenti metodi e oggetti JavaFX senza lettere maiuscole, ed utilizzando i trattini separatori in sostituzione della notazione "camel case". Per esempio la proprietà `textAlignment`

(che impostiamo mediante il suo metodo setter `setTextAlignment()`) diventa `-fx-text-alignment`.

> **Il reference per il CSS da utilizzare con JavaFX si può trovare all'indirizzo:**
> **https://docs.oracle.com/javase/8/javafx/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 S.8.

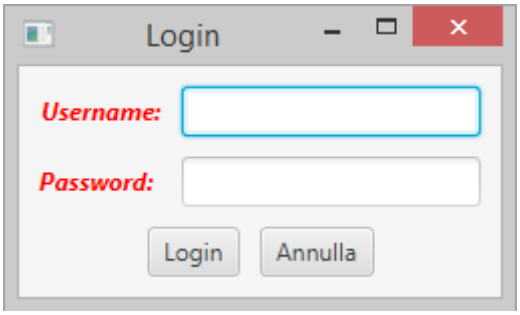


Figura S.8 - Login con FXML e CSS.

S.5.2 CCS e file JavaFX

Per applicare il nostro file CSS ad un file JavaFX, 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 S.9.
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

Figura S.9 - Motore di ricerca con CSS.

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 risultato (sicuramente migliorabile) è visualizzato in figura S.10.

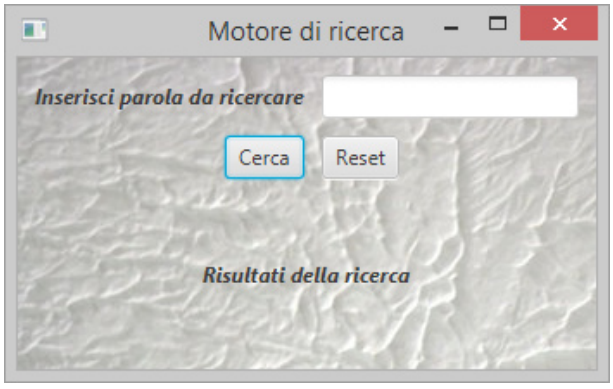


Figura S.10 - Motore di ricerca con immagine di sfondo.

È anche possibile applicare il CSS in modo programmatico, ovvero, invece di associare un file CSS, si possono specificare le istruzioni CSS all'interno di stringhe. Per esempio per impostare l'immagine di sfondo anche ai pulsanti, basterà invocare il seguente metodo:

```
msfButton.setStyle("-fx-background-image: url(\"rock.jpg\")");
```

In questo modo però, si perde il vantaggio di avere la separazione tra i fogli di stile e codice Java.

S.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 occorso 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 è descritta anche la storia e l'evoluzione del modello ad eventi). Quindi in Java i **gestori degli eventi** sono oggetti separati dagli oggetti grafici che utilizzano interfacce come `ActionListener` o classi come `WindowListener`. Mediante 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 scorciatoia più veloce. Con l'avvento delle espressioni lambda e dei reference a metodi, sembra insensato utilizzare le classi anonime nella maggior parte dei casi.
Rispetto a Swing/AWT, JavaFX mantiene lo stesso modello a delega, ma 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.Even` (appartenente al modulo `javafx.base`). Quindi è tutto molto simile a quanto esiste già in Swing/AWT. Se volessimo associare ad un pulsante un'azione, per esempio reimpostare il campo di testo del motore di ricerca alla pressione del pulsante di reset, allora potremmo scrivere:

```
resetButton.setOnAction(event -> text.setText(""));
```

S.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 avan-

zata. 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 un **evento** astratto della libreria, e la JVM invoca il metodo SAM quando viene sollevato l'evento. Ma con JavaFX è possibile sollevare un evento automaticamente, 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 di interfacce funzionali.

S.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 caduta in disuso, che tanto tempo fa era conosciuta come **JavaBeans**.

Il nome JavaBean 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. Era possibile sfruttarli anche all'interno di tool grafici che tramite un semplice trascinamento ne definivano le gerarchie e le associazioni. Questi tool (tra cui ci piace ricordare l'antenato in cui si è evoluto 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 venivano chiamati i metodi setter.
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 la tecnologia JavaFX esegue 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 `javafx.beans.property.Property`. Quindi 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.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();
    }
}
```

Nell'esempio abbiamo istanziato un'oggetto di tipo `SimpleStringProperty`, referenziandolo con un reference di tipo `StringProperty`. 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 associare un gestore di eventi alla proprietà. Quindi sarà l'oggetto `Property` che scatenerà l'evento.

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 crea-

re 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 S.11.

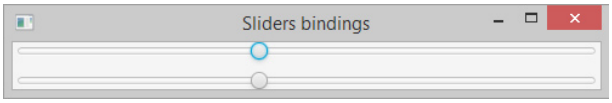


Figura S.11 - Binding di due slider.

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.

S.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 S.12 è possibile vederne l'effetto.

Figura S.12 - Effetto blur.

Oppure possiamo applicare delle **transition**, ovvero delle **animazioni** (package `javafx.animation`, modulo `javafx.graphics`), come per esempio un'anima-

zione 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.

S.9 Componenti grafici avanzati

Come Sun creò l'applicazione dimostrativa SwingSet (ed altre) per mostrare le potenzialità del framework Swing, 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.

S.9.1 Grafico a torta

Un esempio semplice è quello che crea un **grafico a torta** come quello raffigurato in figura S.13. Il codice per generare un grafico come questo è molto semplice:

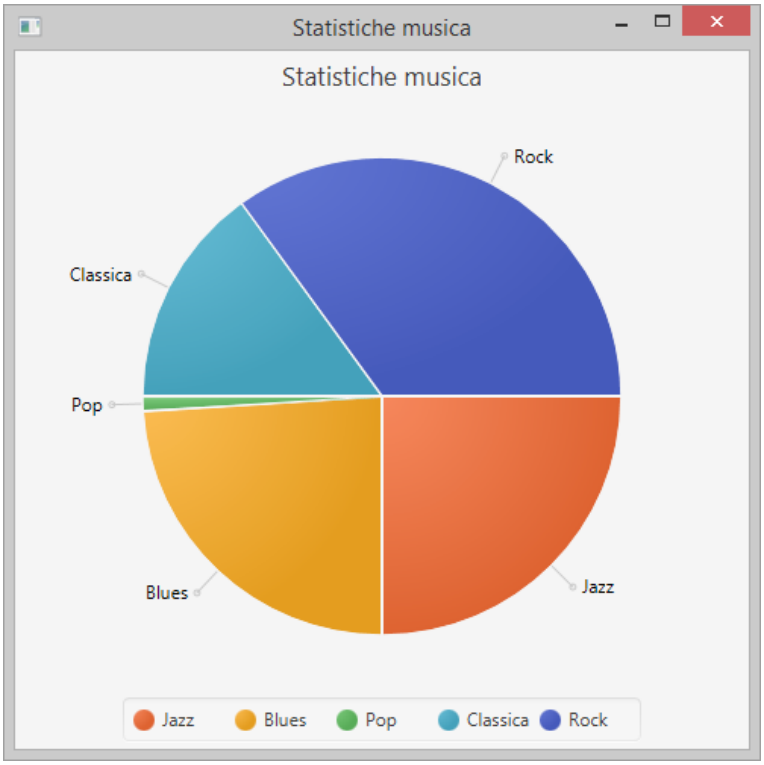


Figura S.13 - Grafico a torta.

```
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");
```

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.

S.9.2 Media Player

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 dell'oggetto `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 si può trovare qui:
https://docs.oracle.com/javase/8/javafx/media-tutorial/playercontrol.htm.

S.9.3 Browser

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 la 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!

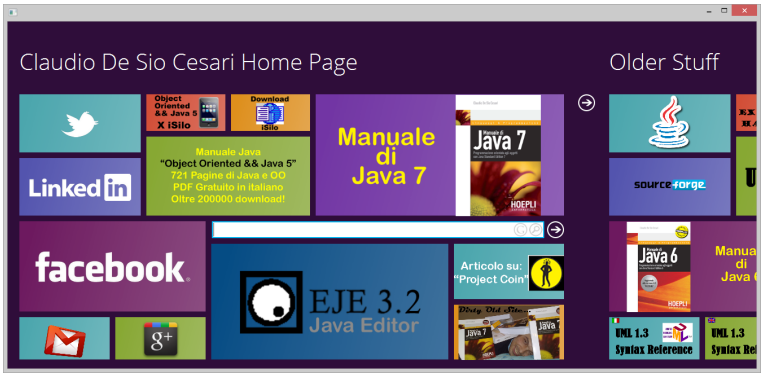


Figura S.14 - Browser con http://www.claudiodesio.com.

Riepilogo

In quest'appendice abbiamo introdotto la tecnologia Java standard per creare interfacce grafiche, che sostituisce 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 quest'appendice 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 con interfacce funzionali invece dei classici Listener definiti in `java.awt`. Abbiamo anche notato quanto ci saranno utili le espressioni lambda (ed i reference a metodi) 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.

Appendice T

Introduzione a Apache Derby

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

✔ Essere capace di installare e utilizzare le funzionalità base di Apache Derby (unità T.1, T.2).

Sino alla versione 8 di Java, il JDK inglobava il database Java DB. Esso altro non era che il database **Apache Derby** su cui Oracle offriva supporto. Con il JDK 9 Java DB non è stato più inglobato nel JDK, quindi per ottenere una copia di questo RDBMS bisogna scaricarlo all'indirizzo: https://db.apache.org/derby. Si tratta di un database scritto completamente in Java e indipendente dalla piattaforma. Esso può essere utilizzato in due modalità differenti:

- come **database embedded** nella nostra applicazione java (ovvero che può essere inglobato direttamente all'interno delle nostre applicazioni). Essendo scritto in Java può essere utilizzato come parte dell'applicazione distribuita, in maniera del tutto trasparente all'utente. In questa modalità il database sarà accessibile ad un solo client (quello che lo ingloba);
- oppure può essere avviato in **modalità Server**, come un normale database engine. In questo caso è possibile accedere con più client al database.

Nel momento in cui stiamo scrivendo queste righe la versione più aggiornata di Apache Derby è la 10.14.1.0.

T.1 Installazione di Apache Derby

Fino alla versione 8 di Java, Java DB era distribuito insieme al Java Development Kit, e quindi era automaticamente installato sulla nostra macchina. Adesso è necessario installare Apache Derby indipendentemente dal JDK seguendo i seguenti passi.

T.1.1 Download del software

All'indirizzo https://db.apache.org/derby/derby_downloads.html fare clic sul primo link con il numero della versione. Nel nostro caso, facciamo clic sul link "10.14.1.0" (vedi figura T.1).

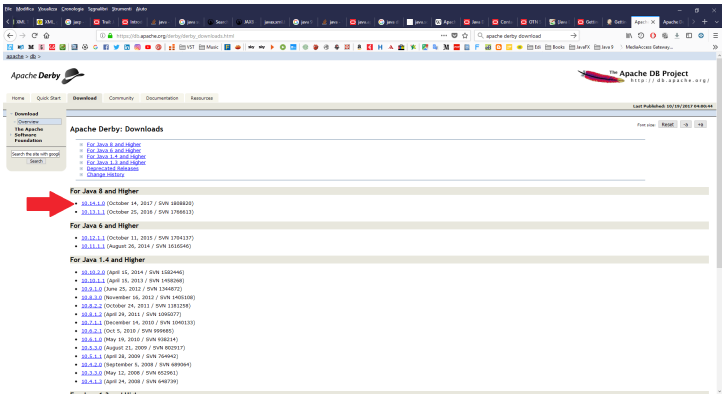


Figura T.1 - Scelta della versione da scaricare (la più aggiornata).

Nella pagina seguente utilizzare il link dei file binari, con il formato più comodo. Noi abbiamo scelto il formato .zip, come è possibile notare in figura T.2.

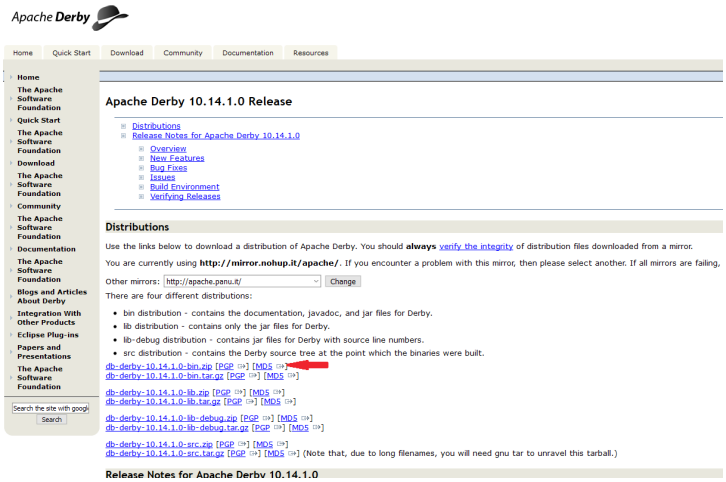


Figura T.2 - Scelta del file da scaricare (il file che contiene i binari col formato più comodo).

In questa pagina è possibile scaricare anche tipologie di file come per esempio i sorgenti.

A questo punto si avvierà lo scaricamento del file zip.

T.1.2 Installazione

Il software non ha bisogno di nessun tipo di installazione particolare, in quanto è scritto in Java. Semplicmente bisogna decomprimere il file scaricato nella cartella che si ritiene più opportuna. Noi per esempio l'abbiamo decompressa nella cartella dei programmi: C:\Program Files\db-derby-10.14.1.0-bin. Nulla ci vieta di posizionare la cartella in qualsiasi altro percorso, e/o di rinominare la sua cartella radice. Per esempio potremmo scaricarla in C:, e rinominare la cartella C:\Program Files\db-derby-10.14.1.0-bin in derby. È possibile osservare il contenuto della cartella di Apache Derby nella figura T.3.

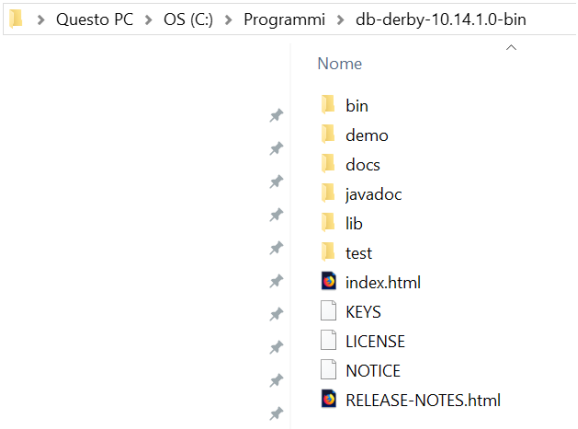


Figura T.3 - Cartella di installazione di Apache Derby.

In particolare si notino le seguenti sottodirectory:

- bin: che contiene gli script di esecuzione e configurazione.
- lib: che contiene i file JAR che rappresentano l'applicazione stessa.

T.1.3 Configurazione

Per rendere l'installazione completamente funzionante bisogna impostare la variabile d'ambiente DERBY_HOME con la directory di installazione di Apache Derby,

che nel nostro caso è C:\Program Files\db-derby-10.14.1.0-bin. Il procedimento per installare la variabile d'ambiente è identico a quello descritto per impostare la variabile PATH nell'appendice B. In figura T.4 è possibile osservare l'impostazione della variabile DERBY_HOME con la cartella di installazione di Apache Derby su un sistema Windows 10.

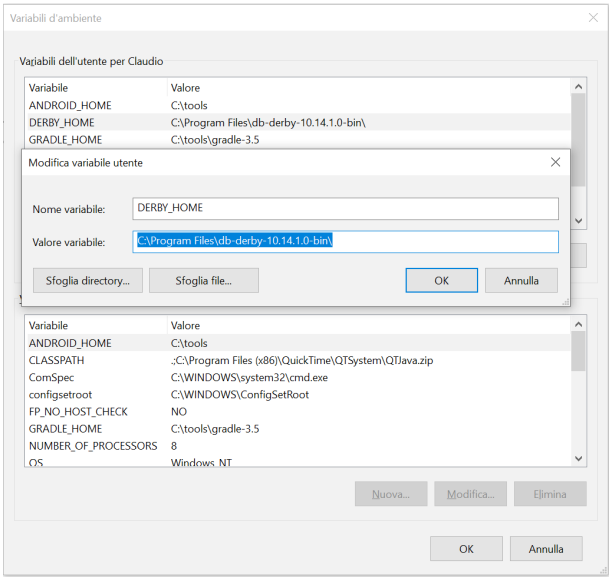


Figura T.4 - Configurazione della variabile d'istanza DERBY_HOME.

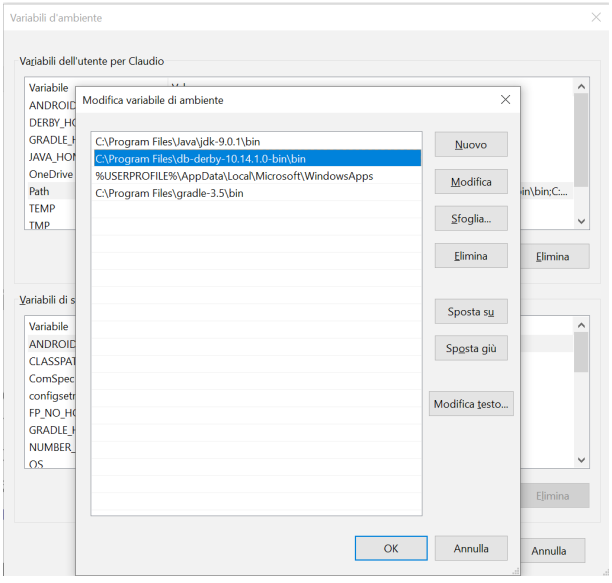


Figura T.5 - Configurazione della variabile d'istanza PATH.

È consigliabile anche aggiungere la cartella C:\Program Files\db-derby-10.14.1.0-bin\bin alla variabile PATH per poter eseguire Apache Derby da qualsiasi posizione del filesystem, senza doversi spostare nella sua cartella di installazione. In figura T.5 è possibile osservare l'impostazione della variabile PATH, aggiungendo la cartella bin di Apache Derby su un sistema Windows 10.

T.2 Esecuzione ed utilizzo di Java DB

Come abbiamo già asserito, Derby può essere eseguito in due modalità: **embedded** e **server**.

T.2.1 Modalità server

Per eseguire Apache Derby in **modalità server**, bisogna utilizzare il file startNetworkServer.bat che si trova nella cartella bin. Possiamo fare clic due volte sul file per eseguirlo direttamente dalla cartella bin (è anche possibile creare un collegamento sul desktop per evitare di navigare nelle varie directory). Eseguendo il file con un doppio click, sarà aperta automaticamente una prompt dei comandi. Lo svantaggio è che se qualcosa andasse storto, tale prompt si richiuderebbe subito, senza darci il tempo di leggere l'eventuale problema. Ma se abbiamo impostato la variabile PATH come indicato nel paragrafo precedente, possiamo eseguire il server anche in maniera alternativa. Basta aprire la prompt dei comandi, ed eseguire il comando startNetworkServer.bat (ricordiamo che da prompt DOS i comandi sono case insensitive, e che è possibile non specificare i suffissi .bat) da una qualsiasi cartella:

```
startNetworkServer
```

Il server sarà avviato in pochi secondi e stamperà in output le seguenti informazioni:

```
Wed Feb 07 01:02:06 CET 2018 : Gestore della sicurezza installato
  con i criteri di sicurezza di base del server.
Wed Feb 07 01:02:11 CET 2018 : Server di rete Apache Derby: 10.14.1.0
  - (1808820) avviato e pronto ad accettare connessioni sulla
  porta 1527
```

A questo punto, come è possibile leggere dalla risposta dell'RDBMS, Apache Derby è pronto ad accettare connessioni sulla porta 1527.

T.2.2 Modalità embedded

Per utilizzare Apache Derby in **modalità embedded**, basta eseguire la nostra applicazione aggiungendo alla variabile CLASSPATH (cfr. appendice E) il puntamento

al file derby.jar (e nel caso derbytools.jar and derbyoptionaltools.jar se ce ne fosse bisogno) che si trova nella cartella lib dell'installazione del database.
Per esempio se vogliamo eseguire il file JDBCApp presente tra gli esempi del modulo 18, bisognerà eseguire da riga di comando la seguente istruzione:

```
java -cp .;%DERBY_HOME%\lib\derby.jar JDBCApp
```

T.2.3 Console interattiva

Per avviare una console interattiva per l'esecuzione di script SQL, eseguire la console interattiva mediante il comando:

```
ij.bat
```

che si trova sempre nella cartella bin. A questo punto sarà possibile creare e modificare un database. Per esempio, con i seguenti comandi:

```
CONNECT 'jdbc:derby:Music;create=true';
CREATE TABLE Album ( AlbumId int, Titolo varchar(20),
  Artista varchar(255), Anno int, PRIMARY KEY (AlbumId) );
INSERT INTO Album (AlbumId, Titolo, Artista, Anno)
  VALUES (1, 'Made In Japan','Deep Purple',1972);
INSERT INTO Album (AlbumId, Titolo, Artista, Anno)
  VALUES (2, 'Be','Pain Of Salvation',2004);
INSERT INTO Album (AlbumId, Titolo, Artista, Anno)
  VALUES (3, 'Images And Words','Dream Theater',1992);
INSERT INTO Album (AlbumId, Titolo, Artista, Anno)
  VALUES (4, 'The Human Equation','Ayreon',2004);
```

abbiamo creato il database popolando un tabella di cui parliamo nell'appendice R. Si noti che ogni istruzione deve essere seguita da un ";" di terminazione prima di premere il tasto INVIO (ENTER).
Questo dovrebbe essere sufficiente per poter svolgere dei semplici esercizi. Nel caso si voglia invece approfondire l'argomento, è possibile leggere la documentazione direttamente al link https://db.apache.org/derby/quick_start.html.

Appendice U

Compilazione di codice obsoleto

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

✔ Comprendere e saper gestire i problemi di compilazione di codice scritto con diverse versioni di Java (unità U.1, U.2, U.3, U.4).

Java 5 introdusse nuove regole nella sintassi (come il ciclo foreach, i generics, gli static import e altro) e addirittura un nuova parola chiave: `enum` (oltre anche a `@interface`). Anche Java 1.4 introdusse una nuova parola chiave: `assert`. Inoltre successive versioni di Java come la 7 con il costrutto "try with resources", e Java 8 con espressioni lambda, type annotation, reference a metodi, etc. hanno ulteriormente alterato la sintassi. Java 10 ancora, ha introdotto la parola chiave `var`, mentre l'introduzione delle parole riservate introdotte in Java 9 (`exports`, `module`, `open`, `opens`, `provides`, `requires`, `to`, `transitive`, `uses` e `with`) non crea problemi nel compilare codice obsoleto, dal momento in cui hanno rilevanza solo all'interno della dichiarazione di un modulo. Per esempio posso utilizzare la parola chiave `module` come reference di una stringa nel seguente modo:

```
public class RestrictedWords {
    public static void main(String args[]) {
        String module = "this string use the reference 'module'";
        System.out.println(module);
    }
}
```

Quindi dobbiamo chiederci: cosa succede quando compiliamo codice che è stato scritto prima della versione che stiamo usando?
Solitamente nulla, il codice viene compilato tranquillamente senza problemi. Il codice può essere compilato con un comando come:

```
javac [opzioni] NomeFile.java
```

Esistono dei casi però dove possono sorgere dei problemi.

U.1 Warning

È possibile imbattersi in qualche warning (cfr. paragrafo 9.5.1) per diverse ragioni. Per esempio nel codice pre-Java 5 non esistevano i generics e le annotazioni. Quindi tutte le collection erano usate come raw type (senza specificare generics) e tutti i metodi che facevano override di un metodo ereditato non erano annotati con l'annotazione standard `@Override`. Questo tipo di codice provoca warning se compilato con compilatori post Java 1.4. Ma i warning non impediscono al compilatore di creare byte code valido, la compilazione ha ugualmente successo. Laddove non è possibile aggiornare il vecchio codice, è comunque possibile andare a disabilitare i warning, o come ultima possibilità semplicemente ignorarli.

U.2 Parole chiave usate come identificatori

In particolare, dato che le versioni 10, 1.4 e 1.5 (meglio nota come versione 5) hanno introdotto nuove parole chiave, potremmo imbatterci in codice obsoleto che fa uso delle parole chiave `var`, `assert` o `enum` come identificatori di variabili. Infatti quando non esistevano queste parole chiave, non era raro imbattersi in classi che dichiaravano i reference di tipo `Enumeration` proprio con l'identificatore `enum`. Se provassimo a compilare una classe del genere con un compilatore 1.5 (o superiore), questo non sarebbe in grado di capire che la parola `enum` deve essere considerata come identificatore invece che come parola chiave, ed otterremmo un errore in compilazione. Lo stesso discorso si applica al codice scritto prima di Java 1.4, che poteva far uso dell'identificatore `assert`, che da Java 1.4 in poi è diventata una parola chiave. Se non è possibile modificare il codice obsoleto allora non ci resta che usare un flag al momento della compilazione per specificare qual è la versione del codice sorgente che si vuole utilizzare:

```
javac -source 1.4 NomeFile.java
```

(cfr. capitolo 9). In questo modo avvertiremo il compilatore di compilare il file

come se fosse un compilatore versione 1.4, ovvero senza tener conto delle nuove caratteristiche introdotte con Java 5 (e quindi senza le nuove parole chiave e le nuove caratteristiche della sintassi).

U.3 Sintassi corrente vs sintassi precedente

Se compiliamo codice che contiene sintassi di Java 5 (static import, ciclo foreach, generics, etc.) usando il flag di cui sopra, otterremo errori in compilazione. Stesso discorso si ripete se usiamo sintassi di versioni più nuove come la versione 7 (try with resources) oppure la 8 (espressioni lambda, etc.).
Quindi se siamo costretti a compilare con un flag `-source` specificando una versione precedente a quella che stiamo usando, allora dovremo limitarci ad usare la sintassi della versione specificata.

U.4 Target

Per quanto riguarda l'esecuzione del codice compilato con il flag `-source 1.4`, se si prevede di eseguire il file con una determinata versione della JVM bisogna specificarla con il flag `-target`, come nel seguente esempio:

```
javac -target 1.4 -source 1.4 NomeFile.java
```

Infatti, mentre l'opzione `-source` dichiara che il codice contenuto nel file (o nei file) da compilare contiene una sintassi valida per la versione 1.4, l'opzione `-target` dichiara la versione della Java Virtual Machine su cui sarà possibile eseguire il bytecode compilato.

Se non vengono specificate queste opzioni, il valore di default sia per `-source` sia per `-target` sarà quello della versione di compilatore utilizzata.

Appendice V

EJE (Everyone's Java Editor)

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

- ✔ Essere capace di installare EJE (unità V.1, V.2).
- ✔ Essere capace di usare EJE (unità V.3, V.4).

EJE (acronimo di **Everyone's Java Editor**, in italiano "editor Java di tutti") è un semplice e leggero editor per programmare in Java. Offre funzionalità di base che si adattano perfettamente a chi deve iniziare a muovere i primi passi con la programmazione Java. Lo sviluppo iniziò soprattutto per fare esperienza sulle interfacce grafiche a partire dal 2001. Inizialmente si chiamava "Color Editor" ed era scritto con AWT. Poi contestualmente alla creazione del primo manuale di Java che scrissi nel 2002 (commissionato da una società di informatica) fu riscritto con la libreria Swing. Sino al 2008 EJE è stato sviluppato solo tramite sé stesso! Poi il progetto si è evoluto molto lentamente per mancanza di tempo, giusto per supportare le nuove versioni di Java. Intanto da quando è ospitato su SourceForge all'indirizzo http://sourceforge.net/projects/eje è stato scaricato oltre 75000 volte da utenti di tutti i continenti. È utilizzato come editor didattico in tante università sparse per il mondo. Per certo sappiamo che è stato utilizzato nelle università di Bergen (Svezia), Adelaide (Australia), Pechino (China), Città del capo (Sud Africa), Buenos Aires (Argentina) e persino alla Carnegie Mellon University (università dove si è laureato James Gosling!), che avanzò anche una richiesta per collaborare per migliorare EJE.

V.1 Requisiti di sistema

Per poter essere eseguito EJE ha bisogno di alcuni requisiti di sistema. Per quanto riguarda l'hardware, la scelta minima deve essere la seguente:

- memoria RAM, minimo 32 MB (raccomandati 1 GB)
- spazio su disco fisso: 888 KB circa

Per quanto riguarda il software necessario ad eseguire EJE:

- piattaforma Java: Java Platform Standard Edition, v1.8.x o superiore
- sistema operativo: indipendente dal sistema operativo. Testato su Microsoft Windows 9x/NT/ME/2000/XP/7/8 (TM), Fedora Core 14, 13, 12, 11, 10, 9, 8, 5, 4, 3, 2 & Red Hat Linux 9 (TM), Sun Solaris 2.8 (TM). Su Mac OS X è stato segnalato il corretto funzionamento da più utenti, con un baco noto riguardante la visualizzazione dei numeri di riga. Dal menu Opzioni è possibile comunque disabilitare la visualizzazione dei numeri di riga su tali sistemi. Non testato su altri sistemi operativi.

V.2 Installazione ed esecuzione di EJE

EJE è un programma multipiattaforma, ma richiede due differenti script per essere eseguito sui sistemi operativi Windows o Linux. Sono quindi stati creati due diversi script di esecuzione:

- eje.bat per sistemi Windows (esiste anche un file eje_win9x.bat per Windows 95/98);
- eje.sh per sistemi Linux.

V.2.1 Per utenti Windows (Windows 9x/NT/ME/2000/XP/7/8)

Una volta scaricato sul vostro computer il file eje.zip:

1. scompattare tramite una utility di decompressione come WinRar o WinZip il file eje.zip;
2. eseguire il file eje.bat (eje_win9x.bat per Windows 95/98) con un doppio clic.

V.2.2 Per utenti di sistemi operativi Unix-like (Linux, Solaris . . .)

Una volta scaricato sul computer il file eje.zip (o eje.rar):

1. decomprimere, tramite gzip o altra utility, il file eje.zip (o eje.rar). Sarà creata la directory EJE.
2. Cambiare i permessi del file eje.sh con il comando chmod. Da riga di comando digitare:

```
chmod a+x eje.sh
```

3. Eseguire il file eje.sh nella directory EJE.

V.2.3 Per utenti che hanno problemi con questi script (Windows 9x/NT/ME/2000/XP & Linux, Solaris)

Da riga di comando (prompt DOS o shell Unix) digitare il seguente comando:

```
java -classpath . com.cdsc.eje.gui.EJE
```

Se il sistema operativo non riconosce il comando, allora non avete installato il Java Development Kit (1.8 o superiore), oppure non avete impostato la variabile d'ambiente PATH con la directory bin **del JDK (vedi appendice B o "installation notes" del JDK).**

V.3 Manuale d'uso

EJE è un semplice e leggero editor per programmare in Java. Esistono tanti altri strumenti per scrivere codice Java, ma spesso si tratta di pesanti IDE che hanno bisogno a loro volta di un periodo di apprendimento piuttosto lungo per essere sfruttati con profitto. Inoltre tali strumenti hanno bisogno di ampie risorse di sistema, che potrebbero essere assenti o non necessarie per lo scopo dello sviluppatore. EJE si propone come editor Java non per sostituire gli strumenti di cui sopra, bensì per scrivere codice in maniera veloce e personalizzata. È stato creato pensando appositamente a chi si avvicina al linguaggio, contemporaneamente alla realizzazione di questo testo.

Le principali caratteristiche di EJE sono le seguenti:

1. Possibilità di compilare ed eseguire file (anche con argomenti) direttamente dall'editor (anche appartenenti a package).
2. Supporto di file multipli tramite il tasto TAB.
3. Colorazione delle parole significative per la sintassi Java.
4. Veloce esplorazione del filesystem tramite un'alberazione delle directory, e possibilità di organizzare ad albero le cartelle di lavoro.
5. Navigabilità completa di tutte le funzionalità tramite tastiera.
6. Possibilità di annullare e riconfermare l'ultima azione per un numero infinito di volte.
7. Utilità di ricerca e sostituzione di espressioni nel testo.
8. Inserimenti dinamici di template di codice (è anche possibile selezionare testo per circondarlo con template di codice) e di attributi incapsulati (proprietà JavaBean).
9. Personalizzazione dello stile di visualizzazione.
10. Possibilità di commentare testo selezionato.
11. Possibilità di impostare messaggi da visualizzare dopo uno specificato periodo di tempo.
12. Popup di introspezione di classi automatico, per visualizzare i membri da utilizzare dopo aver definito un oggetto.
13. Possibilità di aprire la documentazione della libreria standard del JDK in un browser Java.
14. Possibilità di generare documentazione automaticamente dei propri sorgenti mediante l'utility javadoc.
15. Indentazione automatica del codice in stile C o Java.
16. Navigazione veloce tra file aperti.
17. È possibile impostare molte opzioni: tipo, stile e dimensione del font, abilitazione e disabilitazione del popup di introspezione, stile di indentazione, compilazione in base alla versione di Java di destinazione, abilitazione/disabilitazione asserzioni, lingua, stile del look and feel.
18. È possibile stampare i file sorgente.

19. Internazionalizzazione del programma multilingua (Inglese, Spagnolo, Tedesco e Italiano).

20. Supporto di Java versione 8. Attualmente (EJE versione 3.5) è possibile utilizzare anche la versione 9, ma non sono ancora state implementate le funzionalità per compilare in maniera intuitiva.

L'interfaccia che EJE mette a disposizione dello sviluppatore è molto semplice e intuitiva. La figura V.1 mostra EJE in azione. EJE si mostrerà nella versione inglese se eseguito su un sistema operativo in lingua non italiana. È internazionalizzato anche in tedesco e spagnolo.

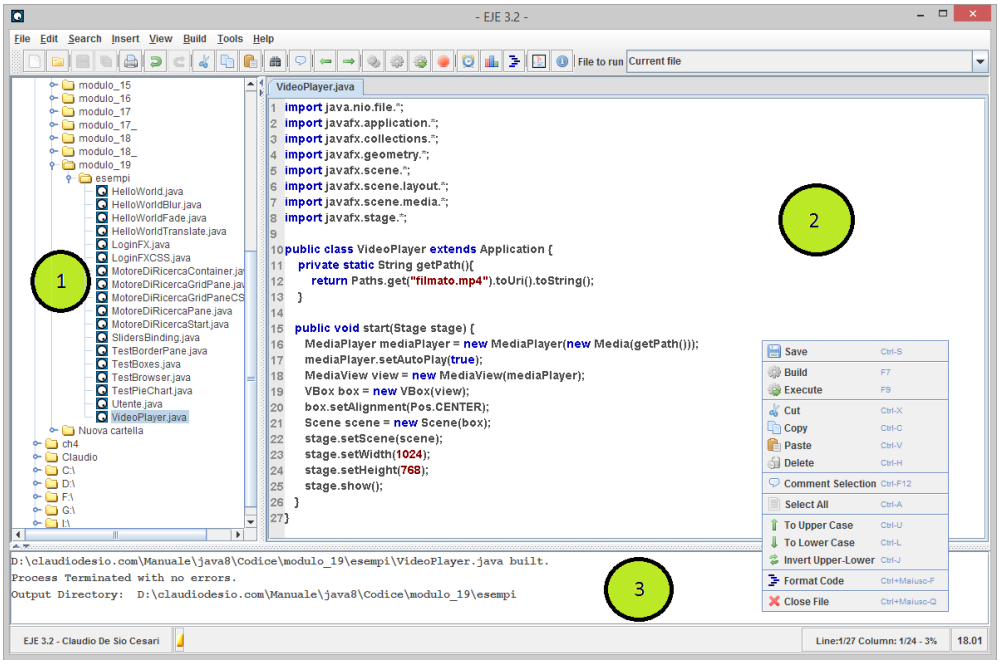


Figura V.1 - EJE in azione.

Il pannello 1 mostra l'alberatura delle cartelle disponibili nel file system. Il contenuto delle cartelle non è visibile a meno che non si tratti di file sorgenti Java. È possibile aprire nel pannello tali file facendo clic su essi. È possibile creare cartelle di lavoro in questo albero come scorciatoie.
Il pannello 2 mostrerà il contenuto dei file aperti.
Il pannello 3 invece mostrerà i messaggi relativi ai processi mandati in esecuzione dall'utente, come la compilazione e l'esecuzione dei file Java.

V.4 Tabella descrittiva dei principali comandi di EJE

Comando	Icona	Dove si trova	Scorciatoia	Sinossi
Nuovo		Menu File, Barra degli strumenti	CTRL-N	Crea un nuovo file
Apri		Menu File, Barra degli strumenti	CTRL-O	Apre un file presente nel filesystem
File recenti...		Menu File		Permette di aprire un file aperto recentemente
Salva		Menu File, Barra degli strumenti, Popup su area testo	CTRL-S	Salva il file corrente
Salva Tutto		Menu File, Barra degli strumenti	CTRL-SHIFT-S	Salva tutti i file aperti
Salva con nome		Menu File		Salva un file con un nome diverso dall'originale
Stampa...		Menu File, Barra degli strumenti	CTRL-P	Stampa il file corrente
Options		Menu File	F12	Apre la finestra delle opzioni
Chiudi File		Menu File, Popup su area testo	CTRL-SHIFT-Q	Chiude il file corrente
Esci		Menu File	CTRL-Q	Termina EJE

Comando	Icona	Dove si trova	Scorciatoia	Sinossi
Annulla		Menu Modifica, Barra degli strumenti	CTRL-Z	Annulla l'ultima azione
Ripeti		Menu Modifica, Barra degli strumenti	CTRL-Y	Ripeti ultima azione
Taglia		Menu Modifica, Barra degli strumenti, Popup su area testo	CTRL-X	Sposta selezione negli appunti
Copia		Menu Modifica, Barra degli strumenti, Popup su area testo	CTRL-C	Copia selezione negli appunti
Incolla		Menu Modifica, Barra degli strumenti, Popup su area testo	CTRL-V	Incolla appunti
Cancella		Menu Modifica, Popup su area testo		Cancella selezione
Seleziona Tutto		Menu Modifica, Popup su area testo	CTRL-A	Seleziona tutto il testo
Rendi Maiuscolo		Menu Modifica, Popup su area testo	CTRL-U	Rende maiuscolo testo selezione
Rendi Minuscolo		Menu Modifica, Popup su area testo	CTRL-L	Rende minuscolo testo selezione

Comando	Icona	Dove si trova	Scorciatoia	Sinossi
Inverti Maiuscolo - Minuscolo		Menu Modifica, Popup su area testo	CTRL-J	Rende minuscolo testo minuscolo e maiuscolo il testo minuscolo
Trova		Menu Cerca, Barra degli strumenti	CTRL-F	Cerca espressioni nel testo
Trova Successivo		Menu Cerca	F3	Cerca la successiva espressione nel testo
Sostituisci		Menu Cerca	CTRL-H	Cerca e sostituisce espressioni nel testo
Vai alla riga		Menu Cerca	CTRL-G	Sposta il cursore alla riga
Template di classe		Menu Inserisci	CTRL-0	Inserisce (o circonda il testo selezionato con) un template di classe
Template di interfaccia		Menu Inserisci	CTRL-1	Inserisce (o circonda il testo selezionato con) un template di interfaccia
Template di enumerazione		Menu Inserisci	CTRL-2	Inserisce (o circonda il testo selezionato con) un template di enumerazione
Template di annotazione		Menu Inserisci	CTRL-3	Inserisce (o circonda il testo selezionato con) un template di annotazione
Template di costruttore		Menu Inserisci	CTRL-4	Inserisce (o circonda il testo selezionato con) un template di costruttore

Comando	Icona	Dove si trova	Scorciatoia	Sinossi
Template di costruttore con argomenti		Menu Inserisci	CTRL-5	Inserisce (o circonda il testo selezionato con) un template di costruttore con argomenti (apre un wizard per la specifica degli argomenti)
Metodo main		Menu Inserisci	CTRL-6	Inserisce (o circonda il testo selezionato con) un template di metodo main
Proprietà JavaBean		Menu Inserisci	CTRL-7	Apre wizard per creare proprietà JavaBean
Template di costante		Menu Inserisci	CTRL-8	Apre wizard per creare una costante
Singleton		Menu Inserisci	CTRL-9	Inserisce un'implementazione del Design Pattern Singleton basandosi sulla classe in cui si trova
If		Menu Inserisci	CTRL-F1	Inserisce (o circonda il testo selezionato con) un template di costrutto if
Switch		Menu Inserisci	CTRL-F2	Inserisce (o circonda il testo selezionato con) un template di costrutto switch
For		Menu Inserisci	CTRL-F3	Inserisce (o circonda il testo selezionato con) un template di costrutto for

Comando	Icona	Dove si trova	Scorciatoia	Sinossi
While		Menu Inserisci	CTRL-F4	Inserisce (o circonda il testo selezionato con) un template di costrutto while
Do While		Menu Inserisci	CTRL-F5	Inserisce (o circonda il testo selezionato con) un template di costrutto do-while
For-Each		Menu Inserisci	CTRL-F6	Inserisce (o circonda il testo selezionato con) un template di costrutto for-each
Try / catch		Menu Inserisci	CTRL-F7	Inserisce (o circonda il testo selezionato con) un template di blocco try/catch
(...)		Menu Inserisci	CTRL-F8	Inserisce (o circonda il testo selezionato con) una coppia di parentesi tonde
[...]		Menu Inserisci	CTRL-F9	Inserisce (o circonda il testo selezionato con) una coppia di parentesi quadre
{...}		Menu Inserisci	CTRL-F10	Inserisce (o circonda il testo selezionato con) una coppia di parentesi graffe.
System.out.println()		Menu Inserisci	CTRL-F11	Inserisce (o circonda il testo selezionato con) un template di costrutto `System.out.println()`
Commenta selezione		Menu Inserisci	CTRL-F12	Inserisce (o circonda il testo selezionato con) un template commento

Comando	Icona	Dove si trova	Scorciatoia	Sinossi
Prossimo file		Menu Visualizza, Barra degli strumenti	F5	Seleziona il prossimo file
File precedente		Menu Visualizza, Barra degli strumenti	F4	Seleziona il file precedente
Barra degli strumenti		Menu Visualizza		Nasconde/visualizza la barra degli strumenti
Barra di stato		Menu Visualizza		Nasconde/visualizza la barra di stato
Scegli cartella di lavoro		Menu Strumenti		Permette di scegliere una cartella di lavoro che verrà aperta nell'albero del pannello 1
Compila		Menu Sviluppo, Barra degli strumenti, Popup su area testo	F7	Compila file corrente
Compila progetto		Menu Sviluppo, Barra degli strumenti	SHIFT-F7	Compila tutti i file aperti
Esegui		Menu Sviluppo, Barra degli strumenti, Popup su area testo	F9	Esegue file corrente
Esegui con argomenti		Menu Sviluppo	SHIFT-F9	Esegue file corrente sfruttando gli argomenti specificati
Interrompi processo		Menu Sviluppo, Barra degli strumenti	ESC	Interrompe processo corrente

Comando	Icona	Dove si trova	Scorciatoia	Sinossi
Sveglia		Menu Strumenti		Permette di impostare un timeout per mostrare un messaggio
Monitor Risorse		Menu Strumenti		Mostra la memoria allocata e usata da EJE
Genera Documentazione		Menu Strumenti		Genera documentazione javadoc del file corrente
Formatta il codice		Menu Strumenti, Barra degli strumenti, Popup su area testo	CTRL-SHIFT-F	Formatta il codice
Guida all'utilizzo		Menu Aiuto	F1	Mostra questo manuale utente
Documentazione Java		Menu Aiuto	F2	Mostra la documentazione della libreria standard Java
Informazioni su EJE		Menu Aiuto	F11	Visualizza informazioni su EJE

Appendice Z

Bibliografia

Obiettivi:

Al termine di quest'appendice il lettore dovrebbe:

- ✔ Essere in grado di consultare un elenco di libri, siti e articoli che l'autore ha ritenuto utili per la stesura di questo libro (unità Z.1).
- ✔ Essere in grado di consultare un elenco di libri, siti e articoli che l'autore consiglia di leggere per proseguire il percorso formativo Java al fine di specializzarsi sulle tecnologie Java (unità Z.2).
- ✔ Essere in grado di consultare un elenco di libri, siti e articoli che l'autore consiglia di leggere per proseguire il percorso formativo Java al fine di specializzarsi su argomenti relativi all'Object Orientation (unità Z.3).

Di seguito è presentato un elenco di libri, siti o articoli che potrebbero interessare al lettore che vuole continuare il suo percorso formativo. Abbiamo diviso la bibliografia in tre sezioni.

1. Nel paragrafo Z.1 sono elencati tutti i libri che sono serviti per scrivere questo libro.
2. Nel paragrafo Z.2 invece, sono consigliate alcune risorse per proseguire in eventuali percorsi formativi finalizzati alla specializzazione delle tecnologie Java più richieste.
3. Nel paragrafo Z.3 infine, sono elencati diversi libri sull'Object Orientation per proseguire in eventuali percorsi formativi finalizzati alla specializzazione delle tecnologie Java.

Tutti (o quasi) i riferimenti sono in lingua inglese.

Z.1 Risorse per Java

Segue una serie di link e testi, che sono stati utili per la stesura di questo libro:

- Paul Deitel, Harvey Deitel "Java 9 for programmers", Prantice hall (2017): un libro completo su Java 9. Uno dei libri migliori in assoluto. Deitel & Deitel sempre molto chiari e precisi.
- Sander Mak, Paul Bakker "Java 9 Modularity", O'Really (2017): fatto molto bene, spiega in maniera impeccabile tutti gli aspetti della modularizzazione in oltre trecento pagine.
- Josh Juneau "Java 9 Recipes", APress (2017): un libro introduttivo a Java 9 basato su esempi. Ottimo solo per un'infarinatura e per le persone che già conoscono Java.
- Kishori Sharan "Beginning Java 9 fundamentals", APress (2017): un ottimo libro introduttivo a Java 9.
- Kishori Sharan "Java 9 Revealed, for early adoption and migration", APress (2017): un ottimo libro solo per le novità di Java 9.
- Joshua Bloch, "Effective Java, third edition", Addison-Wesley (2018): uno dei migliori libri su Java avanzato da uno degli sviluppatori storici di Java. Attesissima nuova edizione a distanza di dieci anni.
- Oracle "Oracle JDK 9 Documentation" (https://docs.oracle.com/javase/9/index.html): in questa pagina è possibile raggiungere i link per conoscere tutte le novità di Java 9.
- Cay S. Horstman, "Java SE 8 for the Really Impatient", Addison-Wesley (2014): è un libro avanzato e sintetico, che riporta solo le novità di Java 8. Horstmann è un autore importante già famosissimo, e questo libro è stato pubblicato qualche mese prima dell'uscita ufficiale della release di Java 8.
- Mala Gupta, "OCA Java SE 7 Programmer I Certification Guide", Manning (2013): un buon libro per affrontare la certificazione Oracle Java 7 Programmer I.
- S. G. Ganesh and Tushar Sharma, "Oracle Certified Professional Java SE 7 Programmer exams 1z0-804 and 1z0-805", APress (2013): altro buon libro sulla certificazione Java, che copre anche l'esame Programmer II.

- Katie Sierra and Bert Bates, "Ocp Java Se 7 Programmer Study Guide", Oracle Press (2013): come il precedente, copre entrambi gli esami Programmer I & II. Gli autori sono gli stessi dei test della certificazione, quindi studiando su questo libro (e soprattutto esercitandosi sui loro test) ci sono ottime possibilità di superare gli esami.

- Philip Heller e Simon Roberts, "The complete Java Certification guide", Sybex (2006): questo libro è molto datato, ma è stato molto importante per capire bene alcune caratteristiche di base di Java.

- Robert C. Martin, "Clean Code: A Handbook of Agile Software Craftsmanship", Prentice Hall (2009): Noto come "Uncle Bob", l'autore Robert C. Martin è un importante esponente del movimento Agile e in questo libro spiega la tecnica "Clean Code" per programmare nella maniera ottimale.

- Doug Lea, "Concurrent Programming in Java: Design Principles and Pattern (2nd Edition)", Addison Wesley publishing (1999): il libro è vecchissimo, ma l'autore è stato il principale architetto dell'originale architettura sulla concorrenza in Java. Consigliato a chi è appassionato di programmazione concorrente.

- Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea "Java Concurrency in Practice", Addison Wesley publishing (2006): una guida pratica sui thread accessibile anche ai neofiti, ma senza le ultime novità della libreria.

- Claudio De Sio Cesari, "Manuale di Java 8", Hoepli Editore (2014): è la versione precedente di questo libro. Rispetto a "Manuale di Java 9", le appendici su JDBC e Java FX, erano gli ultimi due capitoli del libro. Attualmente è stato il libro di programmazione più recensito su Amazon in assoluto, e sicuramente uno dei più venduti.

- Claudio De Sio Cesari, "Manuale di Java 7", Hoepli Editore (2011): Era impostato diversamente rispetto a "Manuale di Java 8". La parte finale era dedicata agli approfondimenti di alcuni argomenti. Questo permetteva ai neofiti di approcciare più gradualmente al linguaggio. Questa struttura era condivisa con "Manuale di Java 6". Con Java 8 è stata scelta una struttura con una curva di apprendimento più ripida, visto che il linguaggio si è arricchito di tante nuove caratteristiche.

- Claudio De Sio Cesari, "Manuale di Java 6", Hoepli Editore (2006): il primo libro della serie. Java 9 condivide ancora oggi con esso la struttura dei primi quattro capitoli.
- Claudio De Sio Cesari, "Object Oriented && Java 5" (2005): è possibile scaricare gratuitamente questo libro dal sito personale dell'autore: http://www.claudiodesio.com. È stato scaricato oltre 450000 volte, ed è costituito da ben 721 pagine, e costituisce anche la base di tutti i manuali pubblicati successivamente. È un manuale completo, ma aggiornato alla versione 5.
- http://docs.oracle.com/javase/specs/: le specifiche ufficiali del linguaggio, per togliersi ogni dubbio.
- https://docs.oracle.com/javase/9/docs/api/overview-summary.html: la documentazione ufficiale di Java 9.
- http://docs.oracle.com/javase/tutorial/: i fondamentali tutorial Oracle.
- http://docs.oracle.com/javase/8/javase-clienttechnologies.htm: tutorial su JavaFX, semplici e pieni di informazioni.
- http://www.claudiodesio.com: il sito personale dell'autore, purtroppo non è aggiornato spesso... ma prima o poi qualche sorpresa arriverà. Da qui sono state estratte alcune appendici di "Manuale di Java 9".
- javarevisited.blogspot.com: blog pieno di post interessanti, brevi e con ottimi esempi.
- http://www.javacodegeeks.com: tanti articoli interessanti su Java.

Z.2 Risorse per tecnologie Java

- Ora che conosciamo il linguaggio Java, è il momento di iniziare ad esplorare le tecnologie che popolano l'universo Java. Oggi le tecnologie più importanti basate su Java sono sicuramente Java Enterprise Edition e lo sviluppo su piattaforma Google Android, che anche non essendo una tecnologia Oracle ufficiale, usa comunque Java (purtroppo versione 7) come linguaggio di programmazione. Ecco una lista di risorse interessanti:
- Arun Gupta, "Java EE 7 Essentials", O'Reilly (2013): un'ottima introduzione all'ultima versione di Java Enterprise Edition.

- http://docs.oracle.com/javaee/7/tutorial/doc/home.htm: il tutorial ufficiale di Oracle.
- Don Brown, Chad Michael Davis, Scott Stanlick, "Struts 2 in action", Manning (2008): il libro più venduto per imparare Struts 2, il framework web più usato in Italia.
- Craig Walls, Ryan Breidenbach, "Spring in action", Manning (2005): il libro più venduto per imparare il framework Spring.
- http://spring.io/docs: la documentazione ufficiale del framework Spring.
- http://www.javaworld.com: tanti articoli interessanti per tutte le tecnologie.
- http://www.theserverside.com: tanti articoli interessanti per le tecnologie enterprise.
- Grant Allen, "Beginning Android", APress (2014): ottimo libro introduttivo su Android.
- Dave MacLean, Satya Komatineni "Pro Android", APress (2014): libro su Android scritto abbastanza bene ma non impeccabile.
- Vladimir Silva "Pro Android Games", APress (2014): spiega come creare giochi con Android.

Z.3 Risorse per Object Orientation

- Questi libri non sono di programmazione ma parlano di metodologie Object Oriented o UML. Alcuni di essi definiscono parti di pura filosofia, quindi stiamo parlando di argomenti molto impegnativi, e relativamente lontani dalla programmazione.
- Martin Fowler, "UML Distilled", Addison-Wesley (2010): un libro indispensabile, coinciso, pratico, pieno di riferimenti e molto diretto.
- Ivar Jacobson, Grady Booch, James Rumbaugh "The Unified Software Development Process" Addison-Wesley publishing (1999): come funziona UP, un libro illuminante ma anche dispersivo.
- Ivar Jacobson, Grady Booch, James Rumbaugh "The Unified Modeling Language Reference Manual" Addison-Wesley publishing (2010): UML dalla fonte.

- Ivar Jacobson, Grady Booch, James Rumbaugh "The Unified Modeling Language User Guide" Addison-Wesley publishing (1999): come sopra.
- Simon Bennet, John Skelton, Ken Lunn "Introduction to UML" McGraw-Hill - Schaum's (2010): discreto libro con un case study significativo.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns" Addison-Wesley publishing (2002): un classico, ma interessante soprattuto dal punto di vista storico. Per collezionisti.
- Mark Grand, "Patterns in Java", Wiley & Sons (2002): indispensabile per chi è appassionato di pattern.
- http://hillside.net/patterns: portale ufficiale della comunità dei pattern.
- Meyer Bertrand, "Object Oriented Software Construction" Prentice Hall (1997): un po' datato, ma assolutamente illuminante.
- Alistair Cockburn, "Responsability Based Model", (RBM) http://alistair.cockburn.us.
- Kent Beck, "Extreme Programming, an introduction" http://www.extremeprogramming.org: una metodologia innovatrice quanto originale.
- Martin Fowler, "Refactoring", Addison-Wesley (1999): un libro molto utile sul refactoring.
- Craig Larman, "Applying UML and Patterns", Prentice Hall (2008): il libro migliore mai letto!