Capitolo 2
Programmazione avanzata

Introduzione

Dopo aver investigato nel corso del capitolo precedente le nozioni di carattere generale relative alla programmazione, con particolare riferimento ai linguaggi basati sul paradigma OO, in questo capitolo l'attenzione è completamente focalizzata sul linguaggio Java. Pertanto, sebbene alcune considerazioni mantengano una valenza generale, la stragrande maggioranza delle regole qui presentate ha un dominio di applicazione limitato al linguaggio Java.

Le nozioni presenti in questo capitolo, come lecito attendersi, presentano un elevato grado di operatività, e includono indicazioni relative a istruzioni il cui utilizzo dovrebbe essere evitato (per esempio `System.exit`), a suggerimenti atti a mantenere la portabilità del linguaggio ("write once run everywhere"), a consigli utili per migliorare le performance, a un efficace programmazione *multi-threading*, etc.

Obiettivi

L'obiettivo di questo capitolo è fornire una serie di direttive operative, *best practice* e quant'altro al fine di supportare il miglioramento della qualità del software Java prodotto. In parti-

colare, in questo contesto sono analizzate anche caratteristiche quali performance, portabilità, programmazione concorrente, etc.

La lettura di questo capitolo dovrebbe permettere di considerare e di identificare tutta una serie di imprecisioni sistematicamente commesse da diversi sviluppatori ed eventualmente di apprendere proficue tecniche di programmazione Java.

In questo capitolo sono affrontate diverse tematiche molto complesse, come per esempio il *multi-threading*, la cui trattazione dettagliata, coerentemente con gli obiettivi del libro, è rimandata sia ad apposite appendici, sia a testi specifici.

Direttive

2.1 Investire nello stile

Uno stile di programmazione lineare, razionale e consistente è requisito fondamentale per la produzione di codice più facilmente leggibile e comprensibile. Logica conseguenza di queste due caratteristiche è un codice più facilmente testabile, mantenibile e perfino riusabile. Ciò è particolarmente importante considerando che:

- una buona percentuale del ciclo di vita del software è utilizzato dal processo di manutenzione (a seconda dello studio considerato, questo fattore può variare da un minimo di 40% ad un massimo di 90%);
- lo stesso codice, durante l'intero ciclo di vita, tende a essere mantenuto da diverse persone; è raro che il codice sia scritto e mantenuto dalla stessa persona;

Pertanto è opportuno investire nello stile del codice partendo dall'utilizzo di una convenzione largamente condivisa. Ciò permette di ridurre i tempi di apprendimento del codice e ne favorisce una comprensione più approfondita, che quindi ne consente un maggiore riuso, una più semplice manutenzione, e così via.

2.1.1 Adottare lo stile di programmazione standard Java

La Sun Microsystem, fin dalle primissime versioni del linguaggio di programmazione Java, ha pubblicato un documento relativo allo standard di programmazione Java (cfr.: [SJAVCC]). Questo include direttive relative al nome delle classi Java, all'organizzazione dei file, all'indentazione del codice sorgente, all'organizzazione delle dichiarazioni, dei costrutti e dei cicli, alle convenzioni sui nomi, etc.

Disponendo di una convenzione standard ben collaudata, documentata accuratamente, condivisa da una larghissima comunità di programmatori distribuiti su l'intero globo terrestre, perché tentare di inventarne un'altra?

2.2 Utilizzare accuratamente le "costanti"

Nel linguaggio Java non esistono costanti nel senso tradizionale del termine. Questo è dovuto al fatto che i progettisti del linguaggio Java hanno deciso di non dotarlo di un precompilatore onde evitare la proliferazione di alcuni tipici abusi presenti nei linguaggi C e C++ (vedere kilometriche introduzioni di istruzioni #ifdef). Pertanto in Java non esiste un meccanismo di sostituzione, a "tempo di compilazione", tra etichette e corrispondenti valori. Come alternativa è possibile definire delle variabili condivise il cui valore non può essere modificato (final) che, pertanto, conviene dichiarare statiche (static).

2.2.1 Non inserire valori *hard-coded* nel codice

Una direttiva base della programmazione, indipendentemente dal linguaggio considerato, consiste nell'evitare di includere valori invariabili direttamente nelle istruzioni del programma. Tale pratica dannosa, indicata tipicamente con i termini di *hard-coding*, è in grado di generare una serie di problemi, ad esempio l'aumento della resilienza alle modifiche, una minore consistenza, e così via.

Pertanto, ogniqualvolta si abbia la necessità di inserire un valore non variabile direttamente nel codice, è necessario valutare se si tratti di un valore che:

1. non cambierà pressoché mai. In questo caso è sufficiente utilizzare una "costante" Java. Per esempio:

```
public static final String NEW_LINE = System.getProperty("line.separator");
```

2. ha buone probabilità di venir aggiornato. In questo caso il ricorso a una variabile statica non costituisce una buona pratica poiché variazioni del valore richiedono di ricompilare il codice, distribuirlo, etc. In questo caso è più opportuno inserire tale valore in un opportuno file di configurazione (property file, file XML, etc.).

3. seppur con scarsa probabilità, potrebbe comunque cambiare. In questo caso, una valida strategia consiste nell'incapsulare i valori in opportuni metodi, affinché un eventuale cambio di strategia risulti assolutamente trasparente. Per esempio: getTextFileExtension().

2.2.2 Valutare il caso di inserire valori costanti in un'opportuna interfaccia Java

Una buona pratica di programmazione consiste nell'includere dentro un'opportuna interfaccia Java le costanti largamente utilizzate in un framework, package o sistema.

Per esempio, si analizzi il caso del listato 1.

Listato 1 – Utilizzo di un'interfaccia per contenere una serie di costanti.

```
public ApplConstants {
```

```
    // Common Strings
    public static final String NEW_LINE  = System.getProperty("line.separator");
    public static final String FILE_SEPARATOR       = System.getProperty("file.separator");
    public static final String PATH_SEPARATOR       = System.getProperty("path.separator");
}
```

2.2.3 Non eseguire l'*hard-coding* dei nomi di file

Questa regola rappresenta una ripetizione di quanto esposto nei precedenti punti; ciò nonostante, si è deciso di enfatizzarla nuovamente per via dell'elevata frequenza con cui viene disattesa. Eseguire l'inglobamento dei percorsi direttamente nel codice può porre seri problemi sia relativi all'istallazione dei sistemi e al riutilizzo del codice, sia alla portabilità del codice. Un caso frequente è relativo all'inizializzazione di stringhe contenenti la concatenazione del percorso del file, incluso il relativo nome. Per esempio: `private String configFile = "C:\mysys\config\config.xml"`.

Ciò crea non solo i tipici problemi dovuti all'*hard-coding*, ma anche di portabilità. Per esempio, nel sistema operativo Unix i percorsi assoluti iniziano con il carattere "/", mentre in Windows, questi iniziano con una lettera identificante l'unità dati.

Per sottrarsi a questi problemi è necessario:

1. evitare l'*hard-coding* dei percorsi ricorrendo alle tecniche illustrate precedentemente;

2. costruire i percorsi utilizzando:
   a. la classe `File(File, String)` per costruire il percorso del fine;
   b. le proprietà di sistema per costruire i percorsi. In particolare: `System.getProperty("file.separator")` e `System.getProperty("path.separator")`

2.2.4 Evitare l'*hard-coding* dei caratteri utilizzati per terminare una linea

Anche questa regola rappresenta un'ulteriore enfatizzazione di quanto enunciato in precedenza ma si tratta di un aspetto troppe volte trascurato.

Il problema consiste nel fatto che il terminatore di linea nei file di testo varia a seconda della piattaforma di riferimento. In particolare, è possibile avere tre differenti convenzioni: "\n" (Windows), "\r" (Unix) e "\r\n" (Apple). Per esempio, la seguente istruzione:

```
System.out.print(i+") \n user=currentUser.getLogin() \n attempts="+failedAttempts);
```

verrebbe mostrata correttamente solo in ambiente Windows

```
1)
user=vettitagl
attempts=2
```

Mentre in ambiente Unix produrrebbe il seguente output:

```
1) \n user= vettitagl \n attempts=2
```

Per evitare problemi di questo tipo è sufficiente utilizzare come sequenza di nuova riga, il risultato della seguente istruzione

```
System.getProperty("line.separator").
```

2.3 Concludere correttamente i programmi Java

Quando si scrivono applicazioni Java è necessario tenere a mente le regole di terminazione. In particolare, un'applicazione Java termina quando terminano tutti i *thread* non daemon (detti anche user *thread*) attivi nella medesima JVM. I *thread* creati dal programmatore, tipicamente, sono non daemon. Qualora l'esecuzione dell'applicazione preveda un solo *thread*, demandato all'esecuzione del metodo il main, l'applicazione termina alla terminazione di tale *thread*.

2.3.1 Non terminare l'esecuzione del programma con l'istruzione System.exit()

L'esecuzione dell'istruzione System.exit() forza l'immediata terminazione di tutti i *thread* in esecuzione nella sottostante JVM e quindi termina anche quest'ultima. Pertanto l'utilizzo di questa istruzione dovrebbe essere evitato.

Uno dei problemi centrali è che l'invocazione di System.exit() non permette ai vari *thread* di terminare in maniera pulita (*gracefully*) e non dà l'opportunità di salvare eventuali dati temporanei, contesti, etc. Tale istruzione, inoltre, limita il riutilizzo del codice e pone seri vincoli all'integrazione del codice in opportuni sottosistemi. Si consideri per esempio il caso in cui un codice sia integrato in un sistema più complesso, oppure in un *framework* di integrazione. In questo caso l'esecuzione dell'istruzione System.exit() provocherebbe la chiusura di tutte le applicazioni in esecuzione sulla medesima JVM. Situazione non sempre auspicabile.

Le direttive della Sun (consultare [2]) stabiliscono "This is drastic. It might, for example, destroy all windows created by the interpreter without giving the user a chance to record or even read their contents. […] Programs should usually terminate by stopping all non-daemon threads; in the simplest case of a command-line program, this is as easy as returning from the main method. System.exit should be reserved for a catastrophic error exit, or for cases when a program is intended for use as a utility in a command script that may depend on the program's exit code."

Un'applicazione dovrebbe terminare sempre in modo controllato (*gracefully*), dando il necessario tempo ai vari *thread* di rilasciare le eventuali risorse bloccate, di registrare la persistenza per eventuali dati temporanei, etc.

2.4 Scrivere correttamente i metodi

Gli oggetti sono entità in grado di eseguire un insieme ben definito di attività che ne rappresentano il comportamento. Queste attività, in termini implementativi, sono rappresentate dai

metodi. Poiché le leggi dell'OO, specie l'incapsulamento, prescrivono che le classi debbano celare al mondo esterno la propria logica interna e in particolare le proprietà strutturali (attributi e relazioni), ne segue che i metodi assumono un ruolo fondamentale: rappresentano l'interfaccia propria delle classi, ossia gli elementi di un oggetto accessibile da parte di altri, il contratto tra la classe che fornisce i servizi e le classi che li utilizzano.

Pertanto, ogni oggetto possiede una propria interfaccia (per così dire intrinseca) costituita da un insieme di comandi (i metodi), ognuno dei quali esegue un'azione specifica. Un oggetto può richiedere a un altro di eseguire una determinata azione inviandogli un "messaggio".

2.4.1 Assegnare ai parametri il tipo più generico possibile

Nel definire la firma dei metodi è consigliabile assegnare ai parametri formali un tipo che sia il più generico possibile. Questa tecnica permette di creare un livello di astrazione tra metodi invocanti e quelli invocati, schermandoli da eventuali variazioni di implementazione dei parametri attuali. In questo modo qualora vari il tipo di un oggetto passato come parametro, le ripercussioni di tale variazione resteranno limitate grazie all'esistenza di metodi che si riferiscono al parametro per mezzo di una conveniente astrazione (tipicamente un'interfaccia). Chiaramente, il tipo deve essere il più generico tra quelli possibili. Come spiegato nella regola successiva, è una pratica sconsigliata definire un tipo generico per poi doverne effettuare il *down-cast* nell'implementazione del metodo per via della necessità di utilizzare metodi specifici di un tipo discendente. Per esempio, è da evitare il caso in cui un oggetto java.util.Stack sia passato come parametro di tipo java.util.List, e poi aver necessità di utilizzare il metodo pop, e quindi dover eseguire un *down-cast* (Stack givenStack = (Stack) aList;), per prelevare l'ultimo elemento inserito e quindi il primo a dover essere reperito. Utilizzare un tipo generico è un principio è necessario soprattutto nella realizzazione di metodi non privati.

Qualora un metodo manipoli una collezione, l'applicazione di questo principio equivale ad asserire di riferirsi a tale oggetto per mezzo della relative interfaccia: List, Set e Map. Da notare che liste e insiemi dispongono di un'interfaccia di livello di astrazione ancora superiore: Collection. Anche il ricorso ad array (si ricordi che in Java gli array sono oggetti) può risultare utile soprattutto per classi di utilizzo generico.

Per esempio, dovendo implementare un metodo della classe Portfolio atto ad aggiungere i relativi strumenti finanziari, potrebbe aver senso implementare una versione che preveda come parametro di input un array di strumenti ed eventualmente eseguirne l'*overloading* utilizzando l'intefaccia List, piuttosto di implementare una versione che preveda un ArrayList.

La firma dei metodi potrebbe assumere le seguenti forme:

```
public void addInstruments(Instrument[] instruments)

public void addInstruments(List instruments)
```

2.4.2 Nell'assegnare il tipo a un parametro, evitare di dover eseguire il *down-casting*

Si effettua un'operazione *down-cast* ogni qualvolta un oggetto trattato come un tipo antenato è forzato a un tipo discendente. Pertanto si passa da un tipo più generale a uno più specia-

lizzato. Giacché solo a tempo di esecuzione è possibile esaminare il tipo dell'oggetto fornito come parametro al metodo, ne segue che il controllo di tipo non può essere effettuato a tempo di compilazione ma solo in esecuzione con la ovvia possibilità di generare errori a runtime.

Si consideri per esempio l'improbabile implementazione di un metodo atto a memorizzare un oggetto properties in uno stream di output.

Listato 2 – Improbabile implementazione di un metodo atto a memorizzare un oggetto di tipo java.util.Properties.

```
public synchronized boolean storeProperties
  (OutputStream out, Dictionary properties) throws IOException {

  BufferedWriter awriter =
    new BufferedWriter(
      new OutputStreamWriter(out, "8859_1")
    );

  if (properties != null) {

    if (properties !instanceof Properties) {
      throw new IllegalParameterException("par. Properties not valid");
    }

    prop = (Properties) properties;

    for (Enumeration e = prop.keys(); e.hasMoreElements();) {
      String key = (String) e.nextElement();
      String val = (String) get(key);
      key    = saveConvert(key, true);

      val = saveConvert(val, false);
      writeln(awriter, key + "=" + val);
    }
  }
  awriter.close();
}
```

Come si può notare, dopo aver eseguito i controlli di rito, si esegue il cast (prop = (Properties) properties;). Quello che si deve evitare, a meno che non sia strettamente necessario, è esattamente quello che accade nel codice presentato: accettare un parametro generico e poi eseguirne il *down-casting*. Questa pratica, come regola generale, è sconsigliabile poiché la firma del metodo permette di dedurre, legittimamente, che il metodo preveda un tipo generico, mentre poi l'implementazione ne richiede uno più specifico. Pertanto, l'implementazione del

metodo richiede vincoli più stringenti di quelli sanciti dalla relativa firma, e quindi, ne viola il contratto.

Il *down-casting* però non è sempre evitabile. Per esempio è frequentissimo nell'implementazione dei metodi equals. A un certo punto, infatti, è necessario passare dal tipo generale Object a quello specifico per poter valutare l'uguaglianza dei vari elementi specifici (cfr. listato 4). Da notare che nel caso dell'equals però, questo è sia necessario per implementare una serie di metodi generici (come per esempio indexOf, lastIndexOf, remove, etc. nelle liste), sia è consentito: non avrebbe senso cercare di confrontare oggetti diversi e qualora si cercasse di fare ciò, il metodo risponderebbe correttamente con un valore false.

Listato 3 – Implementazione del metodo equals nella classe java.util.AbstractList.

```
public boolean equals(Object o) {
  if (o == this)
    return true;

  if (!(o instanceof List))
    return false;

  ListIterator<E> e1 = listIterator();
  ListIterator      e2 = ((List) o).listIterator();

  while (e1.hasNext() && e2.hasNext()) {
    E o1 = e1.next();
    Object o2 = e2.next();
    if ( !(o1==null ? o2==null : o1.equals(o2)) )
      return false;
  }
  return !(e1.hasNext() || e2.hasNext());
}
```

Questa regola, in prima analisi, potrebbe risultare in contraddizione con quella precedente. Da un'analisi più attenta si capisce che non lo è, giacché la regola precedente asserisce di selezionare il tipo più generico tra quelli possibili.

2.4.3 Valutare l'opportunità di restituire un valore null per collezioni

Dovendo implementare un metodo che restituisca una collezione (array, Iterator, List, etc.), nel caso in cui tale collezione sia vuota è spesso conveniente non restituire un valore null, ma un oggetto vuoto. Questa tecnica permette di rendere più agevole il codice del metodo chiamante, eliminando la necessità di effettuare il test per determinare la presenza di un valore null.

Le implementazioni delle collezioni Java, per esempio, nel caso in cui le relative istanze siano vuote, restituiscono comunque un oggetto Iterator.

Nel codice, mostrato al punto precedente, relativo al metodo equals, si può notare che una volta restituiti gli oggetti ListIterator, non è stato necessario eseguire il test per verificare se siano o meno nulli; si può passare direttamente al ciclo while, e quindi il codice risulta più lineare.

2.4.4 Cercare di implementare metodi con un solo punto di ingresso e un solo punto di uscita

Un valido principio di programmazione strutturata, incluso nella programmazione OO, afferma che ogni metodo dovrebbe essere dotato di un solo punto di entrata e un solo punto di uscita. Si tratta di una buona regola che facilita la comprensione e la manutenibilità del codice. In effetti, tipicamente, non è sempre agevole aggiornare un metodo dotato di vari punti di uscita (return), diversi da quelli semplici posti all'inizio del metodo come controllo dei parametri, soprattutto quando la sezione da modificare è quella inclusa tra diversi punti di uscita.

Una legittima eccezione a questa regola è l'implementazione della serie iniziale di test dei parametri di un metodo. In questo caso, eventuali ritorni anticipati sono permessi e anzi tendono a semplificare l'implementazione e la lettura del metodo.

2.4.5 Considerare l'utilizzo di una variabile *result* per i metodi che restituiscono un valore

Molti tecnici, specialmente quelli provenienti dal linguaggio C, trovano molto leggibile l'utilizzo del nome convenzionale result per l'eventuale variabile che memorizza il risultato di ritorno dei metodi.

Listato 4 – Frammento di codice atto a verificare l'uguaglianza di due array di oggetti.

```
public static boolean equals(Object[] o1, Object[] o2) {
  boolean result = true;

  // the same object or both null?
  if (o1==o2) || ( (o1==null) && (o2==null) )
    return true;

  int length = o1.length;
  result = (o2.length == length);

  int i=0;
  while ( (result) && (i <length) ) {
    result = ( (o1 == null) ? (o2 == null) : ( o[i].equals(o2[i]) );
    i++;
  }

  return result;
}
```

2.4.6 Non utilizzare metodi deprecati

I metodi deprecati sono metodi ancora presenti nella versione del JDK che si sta utilizzando, ma dei quali è stata pianificata l'eliminazione nelle versioni future. Pertanto è probabile che il loro utilizzo crei seri problemi di portabilità verso nuove versioni del JDK.

2.5 Implementare attentamente i metodi "accessori" e "modificatori" (get/set)

I metodi accessori (`getXXX()`, `isXXX()`, `hasXXX()`) sono funzioni implementate per accedere al valore degli attributi di un oggetto, mentre i metodi modificatori (`setXXX()`) sono implementati per modificare il valore di questi attributi e quindi variano lo stato del relativo oggetto.
L'utilizzo dei metodi accessori e modificatori è stato, soprattutto in passato, oggetto di appassionati dibattiti. In particolare, vi sono tecnici contrari al loro utilizzo motivato dal fatto che renderebbe il codice meno efficiente e che la loro codifica non sia un brillante investimento del tempo a disposizione. Si tratta di argomentazioni la cui validità è abbastanza opinabile. In effetti, si constata che per asserire che questi metodi generano colli di bottiglia bisognerebbe riuscire a produrre codice estremamente efficiente, che non acceda a risorse I/O, in cui ogni dettaglio presenti un elevato grado di ottimizzazione, etc. Cosa, ovviamente, raramente possibile.
Per quanto riguarda un eventuale migliore l'utilizzo del tempo a disposizione, è sufficiente notare che gli IDE moderni sono in grado di generare automaticamente questi metodi corredati da opportuni commenti JavaDoc.
In questo testo, questi metodi sono consigliati perché aumentano la leggibilità del codice, concorrono a renderlo più robusto, ne semplificano la manutenibilità, e così via.

2.5.1 Insistere sull'utilizzo di metodi accessori/modificatori

Come riportato sopra, l'utilizzo dei metodi accessori e modificatori aumenta la leggibilità del codice, concorre a renderlo più robusto (ogni attributo ha pochi e ben definiti punti di accesso) e ne semplifica la manutenibilità. Pertanto, è sempre opportuno incapsulare gli attributi di una classe assegnando loro visibilità privata e implementando opportuni metodi di accesso e modifica.

2.5.2 Utilizzare i metodi accessori per campi soggetti a validazione

L'utilizzo dei metodi modificatori è necessario in tutti quei casi in cui l'insieme dei valori che un campo può assumere è vincolato. Il ricorso a tali metodi serve per evitare di dover ripetere la stessa sequenza di validazione in diverse parti del codice e per semplificare la manutenzione del codice, qualora si renda necessario dover variare la procedura di validazione.

Listato 5 – Esempio del metodo `setGender` *di una classe* `Person`*.*

```
public void setGender(char aGender) {
  if (aGender == null)
    thorw new IllegalArgumentException("Gender = null");
```

```
    aGender = Character.toUpperCase(aGender);
    if (aGender != 'M') && (aGender != 'F') {
      thorw new IllegalArgumentException("Gender = '"+aGender+"'");
    }

    gener = aGender;
  }
```

2.5.3 Utilizzare i metodi modificatori per campi la cui variazione può influenzare il valore di altri

L'utilizzo dei metodi accessori è assolutamente necessario qualora l'aggiornamento del valore di un campo influenzi il valore di altri campi o richieda l'esecuzione di determinate azioni. In questo caso il ricorso a metodi accessori permette di realizzare un codice più chiaro, più robusto e permette di circoscrivere eventuali effetti "ondulatori" sul valore dei campi.

Listato 6 – Implementazione del metodo setValue *del componente swing* JProgressiveBar. *Questo metodo permette di impostare il valore attuale della barra di progressione e tipicamente richiede il ridisegno del componente.*

```
public void setValue(int n) {
  // updates the underlying model
  BoundedRangeModel brm = getModel();
  int oldValue              = brm.getValue();
  brm.setValue(n);

  if (accessibleContext != null) {
    // forces a repaint
    accessibleContext.firePropertyChange(
      AccessibleContext.ACCESSIBLE_VALUE_PROPERTY,
      new Integer(oldValue),
      new Integer(brm.getValue())
    );
  }
}
```

Un altro interessante esempio è fornito dalla classe astratta Buffer del package NIO.

Listato 7 – Implementazione del metodo limit *della classe* java.nio.Buffer.

```
/**
 * Sets this buffer's limit.  If the position is larger than the new limit
 * then it is set to the new limit.  If the mark is defined and larger than
```

```
     * the new limit then it is discarded. </p>
     *
     * @param      newLimit The new limit value; must be non-negative
     *             and no larger than this buffer's capacity
     * @return     This buffer
     * @throws  IllegalArgumentException If the preconditions on
     *             <tt>newLimit</tt> do not hold
     */
    public final Buffer limit(int newLimit) {
      if ((newLimit > capacity) || (newLimit < 0))
        throw new IllegalArgumentException();

      limit = newLimit;
      if (position > limit)
        position = limit;
      if (mark > limit)
        mark = -1;
      return this;
    }
```

2.5.4 Implementare metodi accessori/modificatori multipli per collezioni

Il ricorso all'utilizzo dei metodi accessori/modificatori, come illustrato precedentemente, presenta una serie di importanti vantaggi, come aumento del livello di incapsulamento, maggiore leggibilità del codice, semplificazione della manutenibilità, e così via.

Tuttavia, nel caso di collezioni, quali array, array dinamici, vettori, tavole hash etc., il ricorso a questi metodi richiede l'implementazione di un insieme più articolato di metodi accessori/modificatori.

Nella tabella seguente è presentato un pattern particolarmente conveniente per l'implementazione di questi metodi.

Da tener presente che metodi di questo tipo sono inclusi in classi diverse da quelle degli elementi che rappresentano. Per esempio il metodo di addOrderLine, atto ad inserire istanze di tipo OrderLine nella classe Order, è, ovviamente, un metodo di quest'ultima. Pertanto nel nome dei metodi get/set è necessario ripetere il soggetto della collezione.

Come esempio si consideri il caso del pattern Observer. In particolare, si consideri un oggetto "osservato" che accetti una lista di oggetti di tipo Listener a cui inviare le segnalazioni di cambiamento del proprio stato.

Vediamo ora una serie di pattern per l'implementazione dei metodi accessori/modificatori di collezioni: al nome del metodo seguono alcuni esempi e una descrizione dello stesso.

setCollection()

Esempi:

```
setObservers(Observer[] observers)
setLineOrders(LineOrder[] lineOrders)
```

Questo metodo esegue due attività molto importanti: azzera la collezione e vi imposta i valori specificati. Questi dovrebbero essere specificati utilizzando il tipo di dati più semplice possibile: array o apposita interfaccia della collezione usata.

```
getCollection()
```

Esempi:

```
getObservers()
getLineOrders()
```

Compito di questo metodo è restituire la specifica collezione. In modo equivalente a quanto asserito per il corrispondente metodo set, anche in questo caso è opportuno riportare il tipo di dati più semplice possibile. Ottimi canditati sono: Iterator, un apposito array, l'interfaccia Collection, e così via.

```
addCollectionElement()
```

Esempio:

```
getObserverElement(Observer aObserver)
getLineOrderElement(LineOrde aLineOrder)
```

oppure:

```
getObserverElement(String observerId)
getLineOrderElement(String lineOrderId)
```

Questo metodo ha l'obiettivo di restituire uno specifico elemento presente nella collezione. Si sarebbero potute utilizzare forme più contratte, tipo getLineOrder, ma queste si prestano a generare confusione con il metodo get della collezione.

```
removeCollectionElement()
```

Esempio:

```
removeObserverElement(Observer aObserver)
```

```
removeLineOrderElement(LineOrde aLineOrder)
```

oppure:

```
removeObserverElement(String observerId)
removeLineOrderElement(String lineOrderId)
```

Questo metodo ha l'obiettivo di rimuovere uno specifico elemento dalla collezione. Anche in questo caso si sarebbero potute utilizzare forme più contratte, tipo removeLineOrder, ma si è preferita la forma più lunga onde evitare confusione.

```
setCollection()
```

Esempi:

```
clearObservers()
clearLineOrders()
```

Compito di questo metodo è rimuovere tutti gli elementi della Collection.

2.6 Utilizzare con oculatezza la classe java.lang.Runtime

> Ogni applicazione Java è fornita di una sola istanza di questa classe i cui metodi le permettono di accedere all'ambiente di esecuzione dell'applicazione. Quantunque questi metodi possano fornire una serie di utili servizi, il relativo utilizzo pone seri problemi di portabilità.

2.6.1 Valutare attentamente l'utilizzo dell'istruzione Runtime.exec

L'istruzione Runtime.exec permette di richiedere al sistema operativo di eseguire in un apposito processo il commando specificato nell'argomento. Sebbene si tratti di un'istruzione indubbiamente potente, il suo utilizzo può porre una serie di problemi di portabilità. Per esempio, si consideri il caso in cui la si utilizzi per eseguire un'applicazione specifica Windows, come per esempio Calc, non disponibile in altri ambienti. Tale utilizzo, chiaramente, limita la portabilità dell'applicazione in ambienti come Unix.

Le direttive Sun (cfr. [2]) raccomandano l'utilizzo di Runtime.exec solo qualora i seguenti criteri siano soddisfatti:

- l'invocazione deve essere un risultato diretto di un'azione specifica dell'utente e pertanto deve essere a conoscenza del fatto che si sta eseguendo un programma diverso, come per esempio un browser Internet;
- l'utente deve essere messo in grado di selezionare, a tempo di esecuzione o direttamente nel processo di invocazione dell'istruzione, il programma da eseguire;

- eventuali problemi nell'esecuzione dell'istruzione devono essere gestiti chiaramente e limpidamente, specialmente se dovuti all'assenza dell'applicazione di cui si è tentato di invocare l'esecuzione;

- l'utilizzo dell'istruzione è indipendente dalla piattaforma e perciò l'applicazione invocata è disponibile per le diverse piattaforme, come l'invocazione del compilatore Java: javac.

2.7 Implementare i metodi Object

La classe java.lang.Object, antenata di tutte le classi Java, definisce una serie di metodi di servizio (per esempio toString, equals, hashcode, clone, etc.) che, sebbene tedioso, in molti contesti è necessario implementare al fine di assicurare il corretto funzionamento delle API Java. Per esempio, il metodo equals è utilizzato dalle collezioni Java per verificare l'uguaglianza di due oggetti, per rimuovere oggetti dalla collezione, etc. Pertanto, qualora si vogliano utilizzare propriamente le collezioni Java è necessario ridefinire il comportamento definito da questo metodo.

2.7.1 Implementare il metodo toString

Ogni classe dovrebbe implementare la propria versione del metodo toString (effettuarne l'overriding). Questo è definito nella classe java.lang.Object e quindi è ereditato da tutte le classi Java. Il suo compito è di riprodurre in una stringa lo stato dell'oggetto. Questa stringa dovrebbe essere sintetica ma descrittiva.

L'implementazione di base del metodo toString è riportata nel frammento di codice del listato 8.

Listato 8 – Implementazione del metodo toString *presente nella classe* java.lang.Object.

```
public String toString() {
  return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

Il metodo toString è molto utile per una serie di motivi, come eseguire il debug delle applicazioni, specie di applicazioni *multi-threading*, scrivere su opportuni file di log lo stato dei vari oggetti, etc. Tipicamente, l'esecuzione di questo metodo non è vincolata da forti requisiti di performance, però in alcuni contesti (per esempio applicazioni *multi-threading* che sfruttano il metodo per effettuare il log dell'applicazione) potrebbe diventarlo. In questi casi è opportuno ricorrere alla classe java.util.StringBuffer o lang.StringBuilder.

2.7.2 Implementare il metodo *equals* per le classi "dati"

Analogamente al metodo toString, anche la versione embrionale del metodo equals è definita nella classe java.lang.Object. Il suo compito è quello di verificare se l'oggetto in questione è, o meno, uguale a quello fornito.

Listato 9 – Implementazione equals *presente nella classe* Object.

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

L'implementazione del metodo deve soddisfare le seguenti proprietà:

- riflessiva: x.equals(x) = true;
- simmetrica: x.equals(y) = true <=> y.equals(x) = true;
- transitiva: x.equals(y) = true and y.equals(z) = true => x.equals(z) = true;
- consistenza: se x.equals(y) = true allora questo deve essere vero per un numero qualsivoglia di invocazioni di tale metodo (a condizione che le due istanze non siano soggette a modifiche).

L'implementazione di questo metodo è fondamentale per tutte quelle classi disegnate quasi esclusivamente per incapsulare dati, come per esempio Value Object e Data Transfer Object. Questo perché il metodo è fortemente utilizzato nelle collezioni standard. Per esempio in Collection.contains, Map.containsKey, Vector.indexOf, etc. etc.

Secondo le direttive Java, se una classe ridefinisce il metodo equals, allora deve ridefinire anche il metodo hashCode.

Listato 9 – Esempio di implementazione del metodo equals *nella classe* Array.

```
public static boolean equals(long[] a, long[] a2) {

  if (a == a2)
    return true;

  if ( a == null || a2 == null)
    return false;

  int length = a.length;
  if (a2.length != length)
    return false;

  for (int i=0; i<length; i++)
    if (a[i] != a2[i])
      return false;
```

```
    return true;
}
```

2.7.3 Implementare il metodo hashCode per le classi "dati"

> La piena comprensione del metodo hashCode e della sua implementazione richiede la conoscenza di alcune nozioni base della teoria dell'hashing. Pertanto, si consiglia di leggere l'Appendice C ad esso dedicata al fine di comprendere più approfonditamente quanto riportato di seguito.

Quanto detto per il metodo equals in merito al corretto utilizzo delle collezioni Java è valido per il metodo hashCode. Per esempio gli oggetti inseriti in un hashtable devono necessariamente ridefinire il metodo hashCode perché questo è utilizzato dai vari metodi della collezione, come per esempio get. Pertanto, qualora questo metodo non fosse definito per una determinata classe, questa potrebbe creare problemi se utilizzata come chiave in collezioni come Hashtable e HashMap.

Per capire appieno l'importanza del metodo hashCode è necessario ricordare che il valore *hash* di un oggetto, utilizzato come chiave in una collezione Hashtable o HashMap, serve per determinarne la posizione dei relativi elementi all'interno di tali collezioni. Poiché però esiste anche il problema delle collisioni (elementi diversi possono generare uno stesso valore *hash*), ne segue che il valore *hash* dell'elemento chiave è utilizzato per accedere alla posizione delle liste di collisione (liste in cui sono memorizzati elementi diversi le cui chiavi danno luogo allo stesso valore *hash*, cfr. listato 11). In effetti, una collezione *hash* non è altro che un array di liste di collisione. Una volta determinata la lista di collisione di un elemento, l'elemento stesso è individuato utilizzando il metodo equals.

Listato 10 – Implementazione del metodo get *della classe* java.util.HashMap.

```
public V get(Object key) {

  Object k = maskNull(key);
  int hash = hash(k);

  int i = indexFor(hash, table.length);

  Entry<K,V> e = table[i];

  while (true) {
    if (e == null)
      return null;
    if (e.hash == hash && eq(k, e.key))
      return e.value;
    e = e.next;
```

```
    }
  }

  static int indexFor(int h, int length) {
    return h & (length-1);
  }

  static int hash(Object x) {
    int h = x.hashCode();

    h += ~(h << 9);
    h ^=  (h >>> 14);
    h +=  (h << 4);
    h ^=  (h >>> 10);

    return h;
  }
```

Come si può notare il valore *hash* della chiave è utilizzato per accedere alla lista dei conflitti (i = indexFor(hash, table.length)) che viene scorsa finché o la lista termina e quindi l'elemento non è presente, oppure finché l'elemento considerato e quello passato come parametro hanno lo stesso valore di hash della chiave *e* questa è esattamente quella cercata ((e.hash == hash && eq(k, e.key))).

Come si può notare, la classe java.util.HashMap implementa un metodo *hash* (hash(Object x)) da utilizzarsi per irrobustire il valore di *hash* restituito dagli elementi forniti. Si tratta di una tecnica atta a sopperire a problemi generati da eventuali metodi hashCode non particolarmente brillanti.

Le proprietà dell'implementazione del metodo hashCode sono:

- consistenza; l'invocazione del metodo, per un dato oggetto, deve ritornare lo stesso valore intero a meno che l'oggetto stesso non sia modificato (in particolare non siano modificati gli attributi utilizzati per la generazione del valore). Chiaramente, se così non fosse, sorgerebbero dei problemi per l'individuazione degli elementi memorizzati in strutture hash;

- uguaglianza; se x.equals(y) = true <=> x.hashCode() = y.hashCode(). Questo implica, tra l'altro, che gli attributi utilizzati nel metodo equals devono essere utilizzati anche nell'implementazione del metodo hashCode.

Da notare che non è richiesto che x.equals(y) = false <=> x.hashCode() != y.hashCode(). Questa proprietà sarebbe molto utile; purtroppo il calcolo delle probabilità insegna che funzioni hash in grado di non generare conflitti, il cui dominio è il tipo int, sono semplicemente non fattibili!

2.7.4 Valutare il ricorso a una classe generica per la generazione di hashCode

Poiché l'implementazione dei metodi hashCode è un'attività assolutamente ricorrente nella programmazione quotidiana, è fortemente consigliato ricorrere all'utilizzo di un metodo generico di generazione dei codici di hash come quello riportato nell'Appendice C dedicata all'hashing.

2.7.5 Implementare il metodo clone

Il metodo clone è utilizzato per restituire una copia (un clone appunto) dell'oggetto in cui è definito. L'implementazione di questo metodo è più complessa di quanto possa sembrare. Infatti, se l'oggetto ha attributi che sono oggetti a loro volta, anche questi dovrebbero essere clonati (in questo caso si parla di "copia profonda", *deep copy*). Questa regola, ovviamente, non è valida per gli oggetti collezione, di cui si accetta la generazione di una copia dell'oggetto collezione che però si riferisca alle medesime istanze degli oggetti memorizzati nella collezione di partenza i quali, quindi, non vengono copiati (in questo caso si parla di "copia superficiale", *shallow copy*).

Listato 11 – Implementazione del metodo clone *nella classe* java.util.ArrayList.

```
public Object clone() {
  try {
    ArrayList<E> v = (ArrayList<E>) super.clone();

    v.elementData = (E[])new Object[size];
    System.arraycopy(elementData, 0, v.elementData, 0, size);

    v.modCount = 0;

    return v;

  } catch (CloneNotSupportedException e) {
    // this shouldn't happen, since we are Cloneable
    throw new InternalError();
  }
}
```

2.8 Porre attenzione alla chiusura degli *stream*

La libreria Java per l'Input e Output (I/O, java.io) è basata sul concetto di flussi (*stream*). Si tratta di un'astrazione utilissima che permette di acquisire e di scrivere informazioni indipendentemente dalla sorgente dati, che può essere un file, una connessione remota, la console, e così via. Inoltre, gli *stream* possono essere tutilizzati congiuntamente a servizi aggiuntivi, come la compressione, la crittografazione, la traslazione, e così via.

2.8.1 Chiudere sempre gli *stream*

Gli *stream* rappresentano risorse la cui corretta cessazione richiede la chiusura esplicita. Questa si ottiene invocando il metodo close. Nelle classi di output l'esecuzione di questa funzione include anche l'esecuzione del metodo di flush.

2.8.2 Non fare affidamento sul metodo finalize

> Ogni classe Java eredita il metodo finalize dalla classe antenata di tutte: java.lang.Object. Non è infrequente il caso in cui questo metodo sia impropriamente utilizzato per rilasciare le risorse utilizzate dall'oggetto.
> Secondo le specifiche Sun, il metodo finalize di un oggetto è invocato dal *garbage collector* nel momento in cui libera la memoria occupata. Pertanto non vi è alcuna garanzia di quando e in che ordine il metodo finalize sarà invocato. Inoltre, questo potrebbe non essere mai invocato fino alla chiusura dell'intera applicazione, evitando quindi che altri oggetti possano accedere alle risorse bloccate da un oggetto dereferenziato. Infine, affidarsi all'esecuzione del metodo finalize potrebbe creare problemi di portabilità. In effetti, la politica utilizzata dal *garbage collector* dipende dal particolare algoritmo adottato per la relativa implementazione, che quindi dipendende, in ultima analisi, dalla specifica implementazione JVM in cui l'applicazione è in esecuzione.

Data l'imprevedibilità intrinseca del metodo, questo non è idoneo per implementare procedure di "pulizia" delle risorse allocate da un oggetto: queste dovrebbero essere rilasciate esplicitamente dopo l'utilizzo o, eventualmente, riconsegnate ad un apposito pool.

L'utilizzo del metodo finalize, in questo senso, potrebbe essere utilizzato solo come precauzione qualora si implementi un *framework* estendibile. La procedura di rilascio delle risorse deve essere affidata a un apposito metodo: close, dispose, etc. che il client dell'oggetto deve chiamare esplicitamente per assicurare il corretto rilascio delle risorse.

2.8.3 Valutare l'utilizzo di oggetti *Buffer*

Per ogni tipologia di flusso (*stream*), le API Java offrono la possibilità di utilizzare una versione dotata di *buffer*. Alcuni esempi di questa tipologia di classi sono: BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter e StringBufferInputStream. Pertanto, ogniqualvolta si abbia la necessità di lavorare con gli *stream*, soprattutto per trasferimenti di dati con file, reti etc., è consigliabile ricorrere all'utilizzo delle versioni dotate di buffer al fine di migliorare le performance.

Da notare che con la versione JDK 1.4, Java è stato dotato di un nuovo package di I/O denominato NIO (New Input Output). In particolare, questo package fornisce una serie di classi per la gestione delle operazione di I/O basate sul concetto di buffer modellato dalla classe astratta java.nio.Buffer e specializzato in diverse forme. In base alle specifiche Sun, si tratta di un'implementazione più efficiente, più scalabile e in grado di gestire l'intero set di caratteri. Pertanto, qualora sia necessario utilizzare oggetti buffer, è consigliato valutare il ricorso alla API NIO.

2.9 Selezionare attentamente le collezioni

Nell'implementazione di classi che manipolano liste di dati è particolarmente importante selezionare correttamente la classe Java delegata a memorizzare e gestire tali collezioni.
Questa selezione, anche per motivi "storici", non è sempre agevole. Ciò essenzialmente per il fatto che le versioni iniziali del linguaggio furono dotate di collezioni *thread-safe*, come Vector e Hashtable, che per questioni di retrocompatibilità, sono state mantenute nelle versioni più recenti. Queste classi, sebbene permettano di realizzare più agevolmente programmi *multi-threading* (la quasi totalità dei metodi sono sincronizzati), fanno pagare, soprattutto in termini di efficienza, la gestione di dell'accesso concorrente (*thread-safety*) anche nei casi in cui ciò non sia richiesto. Si pensi per esempio all'implementazione degli EJB (*Enterprise JavaBeans*), in questo caso il *multi-threading* è gestito dall'application server e quindi ogni esecuzione è eseguita da un opportuno *thread*. Quindi, ulteriori non necessarie sincronizzazioni finiscono per ridurre le performance.
Con la versione Java 2 questo problema è stato risolto introducendo un nuovo insieme di classi collezione, che diventano *thread-safe* solo su richiesta.

2.9.1 Valutare l'utilizzo delle classi ArrayList e HashTable al posto di Vector e Hashtable quando possibile

Le iniziali classi per la gestione delle collezioni come Vector e Hashtable sono state disegnate e implementate includendo la gestione del *multi-threading*.

Pertanto, questa caratteristica (*thread-safety*) finisce per degradare le performance in tutti i casi in cui questa caratteristica non sia necessaria. Pertanto, si consiglia sempre di ricorrere all'utilizzo delle corrispondenti classi introdotte con la versione Java 2 (ArrayList, HashMap e HashSet) in cui la gestione del *multi-threading* è opzionale. In questi casi, qualora, sia richiesta la sincronizzazione dei metodi che modificano le collezioni, è possibile ricorrere a due alternative:

1. sincronizzare i metodi delle classi che incapsulano le collezioni;

2. utilizzare le classi standard disegnate per incapsulare le collezioni dotandole di sincronizzazione. Per esempio, per ArrayList è necessario utilizzare il costrutto `List list = Collections.synchronizedList(new ArrayList(...));`

2.9.2 Scegliere accuratamente la classe da utilizzare

La tabella 2.1 fornisce una sintesi atta a semplificare la scelta della struttura da utilizzare per manipolare una collezione di oggetti secondo i propri requisiti.

Da notare che in tutte le strutture *hash* l'ordine fa riferimento alla sequenza con cui gli elementi sono accessibili nel corrispondente oggetto Iterator.

2.9.3 Valutare l'utilizzo di StringBuffer o StringBuilder

In alcuni contesti in cui è necessario costruire opportune stringhe in diverse fasi dell'esecuzione di un metodo, invece di utilizzare un oggetto stringa, è consigliabile utilizzare la classe

Inter-faccia	Dupli-cati	Classi					JDK 1.0
Set	No	HashSet	…	LinkedHashSet (ordinata – veloce)	…	TreeSet (ordinata – lenta)	…
List	Sì	…	ArrayList	…	LinkedList	x	Vector, Stack
Map	No chiavi Sì oggetti	HashMap	…	LinkedHashMap (ordinata – veloce)	…	TreeMap (ordinata – lenta)	Hashtable, Properties

Tabella 2.1 – Corrispondenza tra le varie strutture dati.

java.util.StringBuffer o, meglio ancora, la versione non sincronizzata introdotta con il JDK 1.5: java.lang.StringBuilder. Il problema della classe String è dovuto al fatto che è immutabile. Pertanto la concatenazione di più stringhe è ottenuta attraverso la creazione di una serie di oggetti stringa intermedi.

Sebbene i compilatori moderni siano in grado di effettuare una serie di ottimizzazioni (soprattutto quando si tenta di costruire una stringa con un solo comando), evitando la generazione di una serie di oggetti intermedi, è sempre opportuno verificare l'utilizzo delle classi StringBuffer o StringBuilder qualora la costruzione richieda diversi concatenamenti eseguiti in tempi successivi.

Listato 12 – Metodo toString della classe java.util.Array.

```
public static String toString(long[] a) {
  if (a == null)
    return "null";
  if (a.length == 0)
    return "[]";

  StringBuilder buf = new StringBuilder();
  buf.append('[');
  buf.append(a[0]);

  for (int i = 1; i < a.length; i++) {
    buf.append(", ");
    buf.append(a[i]);
  }

  buf.append("]");
```

```
    return buf.toString();
}
```

2.10 Implementare attentamente le applicazioni multi-threaded

La programmazione *multi-threading* rappresenta indubbiamente uno degli argomenti più complessi della programmazione. Pertanto, al fine di non appesantire la trattazione e, al tempo stesso, per di fornire ai lettori meno esperti dell'argomento delle nozioni fondamentali, si è deciso di dedicarvi una specifica appendice (Appendice D) a cui si rimanda per spiegazioni di maggiore dettaglio.

I primissimi programmi che ognuno di noi ha scritto erano organizzati secondo una rigida successione di istruzioni che evolveva a partire dal metodo statico main, fino al relativo completamento. Pertanto l'esecuzione del programma era basata sul cosiddetto modello *threaded* singolo, caratterizzato da un flusso di esecuzione assolutamente deterministico. In situazioni di questo tipo, la JVM si fa carico, non appena eseguita, di generare un singolo *thread* di tipo user (detto anche *non-daemon*) incaricato dell'esecuzione del programma a partire dal metodo main(). La JVM genera, inoltre, una serie *thread* di servizio, i *daemon*. Questi operano in background, rispetto al nostro programma, al fine di supportare l'ambiente di esecuzione. Il classico esempio è il *Garbage Collector*. I programmi Java, poi, terminano quando terminano tutti i *thread non-daemon*. In tale evenienza, la JVM si fa carico di bloccare tutti i *thread deamon* e quindi esegue lo *shut-down*.

La JVM, come è legittimo attendersi, gestisce diverse aree di memoria: una serie di *stack* ("pila"), *heap* ("mucchio"), e l'area dei metodi. Ogniqualvolta la JVM genera un nuovo *thread*, questo è fornito di uno *stack* dedicato, destinato ad ospitare variabili locali al *thread* stesso (e quindi non accessibili da parte di altri *thread*), parametri, valori e punti di ritorno dai metodi invocati dal *thread*. Questa area può ospitare esclusivamente tipi primitivi e puntatori a oggetti (*reference*) memorizzati nello *stack*. In Java "collezioni" come String, Array, etc. sono classi e quindi oggetti a tempo di esecuzione (i metodi delle classi sono memorizzati in una particolare memoria, detta memoria dei metodi, mentre gli oggetti, o meglio il loro stato, è memorizzato nello *heap*). Il linguaggio Java, inoltre, prevede il passaggio dei parametri per valore (*by value*) con diverse eccezioni. Per esempio non è possibile (sarebbe assolutamente inefficiente) memorizzare lo stato di un oggetto nello *stack*, pertanto il valore copiato non è l'oggetto stesso ma il relativo puntatore. Gli oggetti, come visto, risiedono esclusivamente nello spazio di memoria *heap*. La JVM ne gestisce uno solo "condiviso" da tutti i *thread*. L'altra area di memoria gestita dalla JVM è quella riservata ai metodi e alle costanti (attributi statici) ed è condivisa da tutti i *thread*. Per una trattazione più approfondita della teoria del multi-threading si consiglia di consultare l'Appendice D.

2.10.1 Preferire l'implementazione dell'interfaccia Runnable

Java dispone di due diverse tecniche per poter definire classi le cui istanze dovranno essere eseguite da appositi *thread*:

1. implementare l'interfaccia java.lang.Runnable;

2. estendere la classe java.lang.Thread.

In entrambi i casi è necessario definire l'implementazione del metodo run(). Qualora si intenda definire esclusivamente il metodo run(), la prima alternativa è da preferire. Questa tecnica è utile anche considerate le limitazioni di Java relative all'ereditarietà singola. Pertanto se una classe eredita da Thread, non può ereditare da altre classi. Da tener presente che sebbene sia sempre possibile simulare l'ereditarietà con un'apposita composizione, alcune volte questa strategia non è utilizzabile proprio poiché è obbligatorio ereditare da un'altra classe, come nel caso di *Applet*.

La principale differenza tra le due strategie è che mentre nel primo caso si ha un solo oggetto eseguito da più *tread*, nel secondo ogni *tread* incapsula anche l'oggetto da eseguire.

Listato 13 – Creazione di un thread tramite estensione della classe java.lang.Thread.

```
class simpleThread extends Thread {

  // Questo meotodo è invocato quando il thread è in esecuzione
  public void run() {
  }
}

// Per creare il thread è necessario eseguire le seguenti istruzioni
Thread thread = new SimpleThread();
thread.start();
```

Listato 14 – Creazione di un thread tramite implementazione dell'interfaccia java.lang.Runanble.

```
class SimpleThread implements Runnable {

  // Questo meotodo è invocato quando il thread è in esecuzione
  public void run() {
  }
}

// Per creare il thread è necessario eseguire le seguenti istruzioni


// 1. Creare l'oggetto di tipo runnable
SimpleThread myRunnable = new SimpleThread();
```

```
// 2. Creare un thread per eseguire l'oggetto runnable
Thread myThread = new Thread(myRunnable);

// invocare il metodo start
myThread.start();
```

2.10.2 Implementare classi thread in modo che terminino programmaticamente

Analizzando programmi concorrenti scritti in Java non è infrequente imbattersi in codici che utilizzano i metodi stop e suspend della classe *thread*. Si tratta di una prassi pericolosa: tali metodi sono stati deprecati poiché rischiosi. In particolare, interrompendo bruscamente l'esecuzione di un *thread* (invocandone il metodo stop) lo si forza a rilasciare tutti i *monitor* precedentemente acquisiti senza permettergli di terminare il task in esecuzione. Tale terminazione avviene per mezzo della propagazione dell'eccezione ThreadDeath. Se un oggetto precedentemente bloccato dal *thread* in questione si trova in uno stato inconsistente durante la terminazione del *thread*, altri *thread* potrebbero utilizzarlo senza aver alcuna opportunità di essere informati circa l'inconsistenza di tale oggetto (che si dice danneggiato, *damaged*). Questa situazione ha le potenzialità di generare comportamenti randomici difficilmente diagnosticabili. A peggiorare la situazione interviene il fatto che l'eccezione ThreadDeath termina i *thread* in maniera "silenziosa" senza fornire all'utente alcun avvertimento del fatto che il programma potrebbe trovarsi in uno stato inconsistente. Situazioni del genere, inoltre, potrebbero generare strani comportamenti solo dopo ore o giorni, rendendone l'individuazione e la diagnosi ancora più problematica.

L'implementazione corretta della terminazione di un *thread* dovrebbe utilizzare un codice simile a quello riportato di seguito.

Listato 15 – Template per la corretta definizione del metodo run*. Da notare che qualora fosse necessario terminare l'esecuzione non appena possibile, anche l'accesso all'attributo* endOfExection *dovrebbe essere sincronizzato.*

```
public class EsampleThread implements Runnable {

  boolean endOfExecution = false;

  /**
   * thread run method
   */
  public void run() {
    boolean endOfWork  = false;


    while ( (!endOfExecution) && (!endOfWork) ) {
```

```
            //perform the work
            endOfWork = someExecution();

        }
    }


    /**
     * Request the thread termination.
     */
    public synchronized void requestStop(){
        endOfExecution = true;
    }

}
```

2.10.3 Non utilizzare il metodo Thread.suspend

Il metodo Thread.suspend non va utilizzato perché è rischioso e come tale è stato accortamente deprecato. In questo caso la deprecazione è avvenuta perché il metodo è intrinsecamente soggetto alla generazione di situazioni di *dead-lock*. Il problema è relativo al fatto che mentre un *thread* è sospeso, nessun altro può accedere alle risorse che questo ha bloccato fintantoché la relativa esecuzione viene ripresa (invocazione del metodo Thread.resume). Ora, se il *thread* incaricato di riavviare quello sospeso dovesse avere la necessità di accedere ad un oggetto tra quelli bloccati da quest'ultimo, ecco generata la situazione di *dead-lock* che si manifesta con la presenza di processi congelati.

Si immagini l'impatto sul sistema di un *thread* bloccato che abbia acquisito una serie di *lock* tra cui uno o più di importanza critica, come per esempio uno che veicola l'utilizzo di una risorsa importante del sistema. Anche in questo caso la situazione può essere risolta in maniera programmatica utilizzando i metodi wait e notify.

2.10.4 Valutare attentamente la manipolazione dei livelli di priorità dei thread

La manipolazione esplicita delle priorità dei *thread* può creare problemi legati alla dipendenza di Java dallo OS (*Operating System*, sistema operativo). In teoria il linguaggio Java definisce dieci diversi livelli di priorità: la classe java.lang.Thread definisce le seguenti costanti (attributi statici):

```
public static final int MAX_PRIORITY  = 10;
public static final int MIN_PRIORITY  =  1;
public static final int NORM_PRIORITY =  5
```

La maggior parte di OS utilizza *scheduler* la cui politica di assegnazione della CPU è basata sulla selezione del *thread* in stato di attesa a più alta priorità. Il problema è che diversi OS

prevedono differenti livelli di priorità. Qualora questi siano superiori o uguali ai dieci livelli (per esempio, Solaris assegna all'attributo di priorità 31 bit, quindi $2^{31} = 2$ Gigabytes) si tratta di risolvere un semplice esercizio di mapping, mentre quando il numero è inferiore (Windows NT dispone di appena sette livelli di priorità), la situazione diviene più problematica (bisogna associare più priorità concettuali a una stessa priorità fisica). A complicare le cose poi intervengono alcuni servizi particolari di specifici OS (per esempio Windows NT) che operano sulla priorità dei *thread*, aumentandola o riducendola, in funzione dell'esecuzione di prestabilite operazioni. Questa tecnica è nota con il nome di *priority boosting* e tipicamente può essere disabilitata attraverso opportune chiamate a codice native, quindi non incluse nel linguaggio Java, tipicamente, effettuate attraverso il linguaggio C.

Pertanto, è sempre consigliabile evitare, per quanto possibile, la gestione esplicita della priorità dei *thread*, e se proprio necessaria, limitarla il più possibile.

2.10.5 Considerare i diversi modelli di esecuzione dei thread

Un importante vincolo da tener presente quando si progettano programmi concorrenti in Java è che, in questo contesto, Java non è proprio completamente indipendente dalla piattaforma di esecuzione. Questa caratteristica fondamentale è ancora ottenibile a spese però di un disegno che contempli, in maniera ridondante, i diversi modelli di funzionamento delle varie piattaforme (il codice deve essere quindi *platform aware*). La questione nodale è che la programmazione *multi-threading* presenta dipendenze strutturali dal sistema operativo, le quali possono essere minimizzate ma non eliminate. Il problema per i programmatori risiede nel fatto che il linguaggio Java non prevede alcun modello *multi-threading* di riferimento; questo è "ereditato" dalla piattaforma di esecuzione. Ciò fa si che se si desidera scrivere programmi concorrenti in grado di essere eseguiti su qualsiasi piattaforma e quindi per qualsiasi modello *multi-threading*, questi devono contenere i meccanismi di funzionamento per i vari modelli.

Il modello più comune, cosiddetto preventivo o a partizione di tempo (*time- slice*), prevede che la CPU sia assegnata ai *thread* per intervalli di tempo (*slices*, "fettine") ben definiti, al termine dei quali, il *thread* è forzato a rilasciare la CPU. In questo caso è il sistema operativo che si fa interamente carico di gestire la concorrenza.

Nell'altro modello, denominato cooperativo, la concorrenza è (quasi) completamente demandata all'applicazione, quindi al programmatore il quale dichiara esplicitamente quale siano i momenti più opportuni in cui un *thread* debba cedere il controllo (invocazione dell'istruzione `yield`), inoltre, potenzialmente, il passaggio del controllo tra diversi *thread*, può avvenire al livello di *user-mode subroutine*, evitando il coinvolgimento di servizi kernel del OS, i quali possono richiedere fino a diverse centinaia di cicli macchina.

Pertanto, i programmi *multi-threading* Java devono essere scritti considerando sia il fatto che il *thread* possa essere forzato a lasciare la CPU per via dell'esaurimento del tempo a disposizione, sia che il rilascio della CPU possa avvenire solo dietro esplicita richiesta da parte dello stesso *thread* (operazione di `yield`).

2.11.8 Porre attenzione ai dead-lock

Una semplice definizione di *dead-lock* ("blocco mortale") asserisce che si tratta di una situazione in cui due o più *thread* interferiscono tra di loro in un modo tale che nessuno può

procedere nella propria esecuzione. Questa situazione si genera quando un *thread* blocca una serie di risorse e che per proseguire abbia bisogno di altre, bloccate da uno o più *thread* che a loro volta, per poter proseguire, hanno bisogno delle risorse bloccate dal primo *thread*.

Sebbene esistano una serie di tool di supporto all'individuazione di *dead-lock*, spesso, si tratta di situazioni difficili da individuare. Ciò nonostante è possibile illustrare una serie di strategie che favoriscono la prevenzione di *dead-lock*. In particolare:

- implementare i thread in modo tale che la procedura per l'acquisizione di lock segua lo stesso ordine: pertanto, qualora un thread trovi bloccata la prima risorsa, automaticamente, non tenta di acquisire le altre;
- inglobare i vari lock in un altro: in questo caso l'acquisizione del lock globale causerebbe l'acquisizione degli altri lock;
- fare in modo che un thread, se ha acquisito un certo numero di risorse ma fallisce ad acquisire le restanti, dopo un paio di iterazioni rilasci tutte le risorse precedentemente acquisite.

Il codice seguente mostra un semplice esempio di deadlock causato da due thread che tentano di acquisire due risorse in senso opposto.

Listato 16 – Classe di esempio di generazione di deadlock.

```
public class SimpleDeadLock implements Runnable {

  private Object resource1;
  private Object resource2;
  private String name;

  public SimpleDeadLock(String aName, Object aResource1, Object aResource2) {

    name = aName;
    resource1 = aResource1;
    resource2 = aResource2;
  }

  public void run() {

    System.out.println("Thread: "+name+" running...");

    synchronized (resource1) {

      System.out.println("Thread: "+name+" locked resource:"+resource1);
```

```
        try {

          System.out.println("Thread: "+name+" sleeping...");
          // il thread viene messo a dormire per 50
          // millisecondi
          Thread.sleep(50);

          } catch (InterruptedException ie) {
            ie.printStackTrace();
          }

          System.out.println("Thread: "+name+" awaking...");

          // al risveglio il thread tenta di effettuare il lock
          // della seconda risorsa
          synchronized (resource2) {
            System.out.println("Thread: "+name+" locked resource:"+resource2);
          }
      }
    }

    public static void main(String[] args) {

      String resourceA = "Risorsa_1";
      String resourceB = "Risorsa_2";

      SimpleDeadLock runnable1 = new SimpleDeadLock("T1", resourceA, resourceB);
      SimpleDeadLock runnable2 = new SimpleDeadLock("T2", resourceB, resourceA);

      Thread tread1 = new Thread(runnable1);
      Thread tread2 = new Thread(runnable2);

      tread1.start();
      tread2.start();
    }
}
```

L'esecuzione del precedente codice genera il seguente output:

```
Thread: T1 running...
Thread: T1 locked resource:Risorsa_1
Thread: T1 sleeping...
Thread: T2 running...
```

```
Thread: T2 locked resource:Risorsa_2
Thread: T2 sleeping...
Thread: T1 awaking...
Thread: T2 awaking...
```

Da tener presente che, qualora si abbia il sospetto di una situazione di *deadlock* o comunque ogni qualvolta si voglia controllare lo stato dei *thread* e monitor di oggetti, è possibile richiedere il *thread and monitor dump* premendo CTRL + BREAK in Windows e CRTL + \ in Solaris.

2.11 Porre attenzione a possibili accessi concorrenti

Disegnare attentamente gli accessi concorrenti a risorse condivise è rilevante in tutti quei casi in cui si voglia implementare un'applicazione *multi-threading*, ossia quando l'applicazione utente voglia gestire diversi *thread*. Da notare che la JVM è *multi-threading* indipendentemente dalla modalità con cui si implementano i programmi che vi dovranno funzionare. Infatti, oltre al *thread* dedicato all'esecuzione del metodo main del programma applicativo, ne esistono altri (denominati *daemon*) demandati alla gestione di una serie di servizi di base, come il *garbage collector*.

La selezione del livello di *lock* da utilizzare, frequentemente, rappresenta un argomento delicato nella progettazione e implementazione di applicazioni concorrenti. Sebbene da un lato una strategia conservatrice potrebbe evitare una serie di problemi tipici di errata sincronizzazione (corruzione dei dati, *race conditions*, comportamenti inattesi in casi non spiegabili, etc.), dall'altro porterebbe alla generazione di un codice non particolarmente efficiente in cui il livello di concorrenza sia piuttosto ridotto e, in alcuni casi addirittura alla generazione di situazioni di *dead-lock*.

2.11.1 Non sincronizzare metodi rientranti

I *thread* Java dispongono di un'area di memoria riservata per il proprio *stack*. Come tale non è accessibile da parte di altri *thread*. Tra gli altri dati (come per esempio l'indirizzo dell'istruzione da eseguire dopo l'esecuzione del metodo corrente), in questa area sono memorizzate le variabili locali dei metodi, i valori dei parametri attuali dei metodi e l'eventuale valore di ritorno. Si ricordi che in Java i parametri sono passati *by value*… o quasi. Chiaramente, nello *stack* non è possibile memorizzare oggetti, ma solo i relativi riferimenti e pertanto, qualora sia necessario passare un array o un altro oggetto, ciò che viene copiato nello *stack* è il relativo indirizzo. In ogni modo, se all'interno di un metodo viene creato un oggetto (per esempio una stringa), il relativo riferimento è visibile unicamente dal *thread* che in quel momento sta eseguendo il metodo stesso. Metodi che utilizzano esclusivamente i parametri forniti e variabili locali sono definiti rientranti e come tali possono essere eseguiti, contemporaneamente, da diversi *thread* senza aver bisogno di alcuna sincronizzazione.

2.11.2 Sincronizzare l'accesso a dati condivisi

Le classi Java disegnate per funzionare in modalità *multi-threading* devono essere progettate per essere *thread-safe*: devono funzionare correttamente qualora i relativi metodi siano ese-

guiti simultaneamente da diversi *thread*. A tal fine è necessario identificare le aree ad accesso esclusivo (aree che se accedute da più *thread* contemporaneamente potrebbero dar luogo a comportamenti errati). Nell'accesso alle risorse di I/O queste aree sono già state analizzate e risolte dall'implementazione delle API Java. Per esempio, la classe java.io.File prevede una serie di metodi sincronizzati, quali:

```
private synchronized void writeObject(ObjectOutputStream s) throws IOException;

private synchronized void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException

public static File createTempFile(String prefix, String suffix, File directory)throws IOException
```

Il metodo createTempFile possiede una sincronizzazione nell'interno della relativa implementazione.

Pertanto, nella stesura del codice è necessario identificare correttamente e gestire conseguentemente le aree dati condivise da più *thread*. Questo è necessario perché la JVM gestisce un'unica area dati (denominata *heap*) condivisa da tutti i *thread* (anche i daemon come il Garbage Collector) destinata a memorizzare tutti gli oggetti (o meglio il valore dei relativi attributi) gestiti dall'applicazione.

Alcuni esempi di dati condivisi sono quelli scritti da un *thread* che poi dovranno essere letti da altri, o viceversa dati da acquisire prodotti da altri *thread*, dati di sincronizzazione, etc.

Una volta identificate le aree e i dati da proteggere è necessario stabilire il tipo di *lock* richiesto… Questo però è argomento di un'altra regola.

2.11.3 Valutare attentamente la necessità di sincronizzare i metodi

L'utilizzo della parola chiave synchronized, come lecito attendersi, ha un impatto sulle performance, sia in termini di latenza (*latency*, tempo impiegato per svolgere un determinato lavoro), sia in termini di *throughput* (numero di lavori eseguito nel medesimo arco temporale). Tutti gli oggetti Java hanno, potenzialmente, un oggetto associato, denominato *monitor*, è necessario serializzarne l'accesso da parte di diversi *thread*. Questo però è effettivamente utilizzato solo quando un oggetto dichiara aree sincronizzate per mezzo della parola chiave synchronized.

Nelle versioni iniziali della, JVM (*Java Virtual Machine*), alcune statistiche evidenziavano che l'impatto di aree sincronizzate poteva giungere fino ad un fattore 50. Le cose sono molto migliorare con le versioni più recenti. Resta, tuttavia, il fatto che oltre a ridurre le performance, può generare colli di bottiglia e avere un effetto negativo sulla scalabilità dell'intera applicazione. Pertanto, è necessario sincronizzare solo quando sia veramente necessario e selezionare il livello opportuno di *lock*.

> La perdita di prestazioni dovuta all'utilizzo della parola chiave synchronized è facilmente comprensibile considerando la politica con cui le JVM, gestiscono la memoria in presenza di thread (le relative specifiche fanno parte del JMM, Java Memory Model). In particolare, ogni thread è fornito con un'apposita cache la cui politica di aggiornamento dei valori è fortemente

influenzata dalla presenza di aree sincronizzate. In assenza di queste, un thread è lasciato libero di evolvere accedendo alla copia "locale" dei valori delle variabili memorizzate nella propria cache. Pertanto (sempre secondo quanto stabilito dal JMM) i thread sono autorizzati ad avere valori diversi relativi alla stessa variabile. La situazione cambia notevolmente in presenza di aree sincronizzate. In questo caso le direttive richiedono che un thread invalidi la propria cache, e quindi la aggiorni con i valori presenti nella memoria "principale", non appena acquisito un lock (ingresso in un'area sincronizzata), e che riporti tutte le modifiche effettuate nella memoria principale, appena prima di rilasciare il lock (uscita dall'area sincronizzata). Pertanto è facile comprendere come ripetute richieste di sincronizzazione da memoria principale verso la cache del thread e viceversa, portano a una riduzione delle performance.

2.11.4 Considerare l'utilizzo di classi di sincronizzazione

Non è infrequente il caso di implementare delle classi e di non sapere a priori se dovranno o meno funzionare in un contesto *multi-threading*, o meglio ancora, di implementare classi che dovranno funzionare in entrambi gli scenari. La strategia più frequentemente utilizzata consiste nell'utilizzare un approccio sicuro e quindi all'introduzione blocchi a mutua esclusione. Questo, se da un lato risolve il problema, dall'altro finisce per penalizzare tutti i contesti *single-threaded*. Un'ottima strategia, di frequente impiego, consiste nell'implementare la classe senza aree sincronizzate e quindi adatta per un ambiente *single-threaded*, e di utilizzarla un'altra che ne incapsuli i metodi per un funzionamento *multi-threading*. Un po' come avviene con le classi `Collection` che divengono thread-safe con l'esecuzione del comando `Collections.synchronizedXXX()`.

2.11.5 Fare attenzione all'atomicità dei tipi long e double

La proprietà di atomicità applicata agli attributi di un programma Java garantisce che un *thread*, in un qualsiasi intervallo di tempo, acceda o al valore iniziale o quello finale di uno specifico attributo. In altre parole, assicura che non sia possibile da parte di un *thread* accedere a valori aleatori generati da diversi *thread* intenti a modificare contemporaneamente il valore della stessa variabile.

Questa proprietà è garantita per tutti i tipi base ad eccezione di `long` e `double` (non dichiarati `volatile`). Questo perché la relativa implementazione utilizza una dimensione di 64bit che spesso è trattata come due *word* da 32bit. Pertanto ciascuna operazione di lettura e scrittura richiede una doppia operazione che, in determinate condizioni, può aver luogo con una distanza temporale (relativamente) significativa. Ciò, pertanto, potrebbe portare al verificarsi della situazione in cui due o più *thread* modifichino e/o leggano la stessa variabile all'unisono ottenendo un valore errato dovuto all'intrecciarsi delle operazioni da 32bit.

Pertanto, l'utilizzo di variabili o attributi di tipo `long` e `double` (non dichiarati `volatile`), in un ambiente *multi-threading* va reso esplicitamente atomico. L'atomicità è condizione *necessaria ma non sufficiente* per la programmazione concorrente (per esempio non garantisce l'accesso al valore più recente della variabile).

La JVM garantisce l'atomicità di attributo o variabile di tipo `long` e `double` dichiarato `volatile`.

2.11.6 Fare attenzione al tipo di lock richiesto

Per ogni blocco di codice Java è possibile selezionare una serie di protezioni per l'accesso concorrente, quali: nessuna, *lock* legato alla singola istanza, e *lock* statico (associato alla classe).

L'acquisizione del *lock* istanza da parte di un *thread* (ingresso in un'area synchronized) blocca eventuali altri *thread* che ne desiderino eseguire metodi sincronizzati. Ciò chiaramente non blocca tali *thread* dall'invocazione di metodi non sincronizzati o sincronizzati a livello di classe.

Listato 17 – Metodo isEmpty della classe java.util.Vector.

```
/**
   * Tests if this vector has no components.
   *
   * @return <code>true</code> if and only if this vector has
   *     no components, that is, its size is zero;
   *     <code>false</code> otherwise.
 */
public synchronized boolean isEmpty() {
   return elementCount == 0;
}
```

Come regola generale, è necessario sincronizzazione, secondo la tecnica desiderata, i metodi che modificano gli attributi delle classi le cui istanze sono condivise. Qualora, poi, sia necessario che i *thread* acquisiscano sempre il valore più recente possibile, è necessario sincronizzare anche i metodi accessori. Si consideri una classe che contenga i dati, aggiornati in tempo reale, dei prezzi di strumenti finanziari che, tipicamente, sono aggiornati diverse volte al secondo. In questo caso è necessario che ogni *thread* acquisisca il valore aggiornato e, quindi, anche i metodi accessori devono essere opportunamente sincronizzati.

Non è infrequente il caso in cui *lock* a livello di istanza presentino un'eccessiva portata e che quindi siano la causa di colli di bottiglia. In questi casi è necessario ricorrere a *lock* più granulari. Si consideri l'esempio, di un oggetto che manipoli diverse strutture dati. In questo caso la presenza di metodi sincronizzati potrebbe risultare non efficiente: l'acquisizione del *lock* da parte di un *thread* per modificare una struttura dati, blocca automaticamente l'accesso agli altri *thread*, magari interessati a manipolare un'altra struttura dati, indipendente dalla precedente. In questi casi, un'interessante alternativa consiste nel ricorre all'utilizzo di oggetti fittizi di cui utilizzare il relativo *monitor* per l'accesso a specifiche sezioni critiche. Per esempio:

Listato 18 – Esempio di codice per l'utilizzo di lock più selettivi.

```
public class TestFinerLock {

  // Dummy object used for locking purposes
  Object lockWrite = new Object();
```

```
    public void write() {

      // non critical implementation

      synchronized ( lockWrite ) {
        // write data in the shared objects
      }

      // do something else
    }
  }
```

I *lock* statici, infine, in maniera del tutto analoga a quanto riportato pocanzi, generano il blocco di *thread* che desiderino eseguire metodi statici sincronizzati appartenenti alla stessa classe, mentre non bloccano *thread* che desiderino eseguire metodi non sincronizzati o sincronizzati al livello di istanza.

2.11.7 Porre attenzione all'utilizzo della classe Iterator per gli accessi concorrenti

Le nuove collezioni Java restituiscono oggetti di tipo Iterator implementati secondo la strategia *fail-fast*. Ciò significa che se, dopo la creazione dell'oggetto Iterator, la corrispondente lista subisce delle modifiche strutturali (aggiunta, rimozione di elementi e variazione e variazione delle dimensioni), non ottenute attraverso l'esecuzione dei metodi propri dell'Iterator, questo lancia un'eccezione di modifica concorrente (ConcurrentModificationException). Pertanto, qualora si verifichi una modifica strutturale concorrente, l'oggetto Iterator fallisce immediatamente e in maniera pulita (da cui il nome della strategia), evitando di correre il rischio di generare problemi di comportamento non deterministico difficili da individuare.

Riassumendo, problemi di modifica concorrente relativi agli Iterator, si verificano nel caso in cui mentre un *thread* sta scorrendo un Iterator, un altro modifica la lista sottostante. Inoltre, non è sempre possibile garantire questo comportamento soprattutto in presenza di metodi che gestiscono le strutture non sincronizzati. Questa strategia è pertanto implementata secondo il criterio *best-effort*.

Qualora sia necessario navigare gli elementi di una lista in una situazione di forte concorrenza, è necessario ricorrere a una delle seguenti alternative:

1. eseguire un lock della lista sottostante;

2. ottenere una copia "privata" degli elementi della lista.

Chiaramente le due alternative presentano ben definiti pregi e difetti e pertanto si configurano come soluzioni ideali per ben definiti scenari. La prima soluzione tende a ridurre la concorrenza, evitando però problemi di performance dovuti alla copia di elementi e pertanto va utilizzato qualora la concorrenza non sia elevata e sia relativa a strutture dati di dimensioni

considerevoli. La seconda, ovviamente, offre una strategia opposta e pertanto è adatta qualora ci sia un forte parallelismo su strutture dati di dimensioni limitate.