Capitolo 3
I commenti

Introduzione

Obiettivo del capitolo è presentare una serie di tecniche, linee guida e *best practice* finalizzate al miglioramento del livello di documentazione del codice. Il corretto funzionamento del software prodotto dovrebbe costituire un prerequisito irrinunciabile dell'ingegneria del software, ma pur sempre di prerequisito si dovrebbe trattare. Un'elevata qualità del codice, infatti, richiede il soddisfacimento di ulteriori importanti qualità come, per esempio, buona leggibilità e la manutenibilità. Pertanto, un'efficace documentazione è una caratteristica irrinunciabile di qualsiasi software.

Sebbene produrre un'efficace documentazione sia uno degli argomenti fondamentali di studio in qualsiasi corso di programmazione, e pertanto tutti gli sviluppatori, in qualche misura, abbiano studiato la relativa teoria, nella pratica lavorativa non è infrequente imbattersi in codici mal documentati, o, addirittura, non commentati.

Una delle regole fondamentali — frequentemente disattesa — per la produzione della documentazione consiste nel redigerla di pari passo con la produzione del codice. Ciò, oltre a fornire uno strumento di prima verifica della corretta implementazione dei vari algoritmi, facilita la scrittura di commenti significativi ed evita l'odiosa attività, spesso e volentieri delegata al programmatore junior di turno, di scrivere frettolosamente la documentazione a posteriori, magari nel brevissimo intervallo che precede il rilascio del codice.

Il problema sostanziale è che il tipico ciclo di vita di un qualsiasi software prevede che lo stesso sia mantenuto, in diversi stati della sua evoluzione, da personale diverso dagli stessi auto-

ri, spesso non presente nelle fasi iniziali di progettazione. Proprio per questo motivo, il codice deve possedere la proprietà di poter essere compreso, in ogni momento, sia dallo stesso autore sia da persone estranee al suo sviluppo. Purtroppo, questa proprietà è spesso disattesa tanto da verificarsi spesso la situazione paradossale di assistere ad autori di software che per via di una documentazione carente o per la mancata applicazione di standard di qualità si trovino in difficoltà nel comprendere, e quindi nel modificare, parti di programma sviluppate da loro stessi.

Il monito è che software non ben documentati, e quindi poco leggibili e difficilmente manutenibili, presentano notevoli problemi di riutilizzo e tendono a far nascere il desiderio di riscrittura nel personale addetto alla relativa manutenzione: non vi è alcuna virtù in un software difficilmente modificabile.

Come lecito attendersi, il linguaggio preso come riferimento è Java, quantunque, la maggior parte delle direttive proposte mantenga la sua validità anche nell'ambito di diversi linguaggi di programmazione. Questo è il caso anche per la documentazione JavaDoc: nessuno vieta né di utilizzarne un'appropriata versione per linguaggi diversi da Java, né di scriverne una simile.

Obiettivi

L'obiettivo di questo capitolo è fornire una serie di direttive atte a migliorare la documentazione del codice prodotto; pertanto la proprietà del software su cui si focalizza l'attenzione è, ancora una volta, la leggibilità. Come largamente discusso nel Capitolo 1, si tratta del prerequisito irrinunciabile per la manutenibilità del codice.

Sebbene tutti concordino sull'importanza di una documentazione chiara e completa, non è infrequente imbattersi in codici mal documentati o non documentati per nulla. Codici non facilmente leggibili e quindi non comprensibili in maniera facile, difficilmente incontreranno l'approvazione del personale addetto alla loro manutenzione.

Gran parte degli argomenti trattati in questo capitolo prevedono come requisito una buona conoscenza e padronanza dell'utility Java per la produzione automatica della documentazione: JavaDoc. Pertanto, sebbene un'approfondita trattazione di tale argomento esuli dai fini di questo libro, nell'Appendice A è riportata una utile e concisa presentazione dell'utility JavaDoc inclusi i relativi tag.

Direttive

3.1 Investire nella documentazione

Sebbene questa regola sia universalmente accettata e quindi possa sembrare inutile da includere in questo contesto, la realtà quotidiana ci insegna come non sia affatto raro imbattersi in codici mal documentati o non documentati affatto. Codice difficilmente leggibile e comprensibile diviene complesso da mantenere e ciò prelude alla necessità di riscriverlo (anche se magari non sarebbe necessario).

3.1.1 Produrre la documentazione contemporaneamente all'attività di codifica

Si tratta probabilmente di una delle principali regole per l'attività di documentazione e probabilmente anche di una delle più disattese. Scrivere la documentazione in linea con la programmazione permette di ottenere una serie di vantaggi, quali:

- fornisce un primo strumento di verifica semi formale del codice che si sta scrivendo. La necessità di redigere in linguaggio naturale la spiegazione dell'algoritmo oggetto di implementazione stimola, implicitamente, a valutarne sia la validità sia la correttezza della codifica; inoltre, qualora risulti difficile commentare porzioni di codice, potrebbe essere il segnale di allarme di un algoritmo errato, contorto o non ben scritto;

- permette di illustrare efficacemente le decisioni prese nello stesso momento in cui vengono prese;

- evita stressanti e frettolose attività di documentazioni nel breve periodo che precede il rilascio del codice.

La scrittura a posteriori dei commenti è frequentemente un'attività tediosa, e pertanto, spesso assegnata a programmatori junior molte volte estranei alla progettazione iniziale. Ad aggravare la situazione interviene il semplice fatto che, tipicamente, documentare un codice privo di adeguati commenti è un compito complesso. La logica conseguenza è che, per rispettare i tempi di consegna, si finisce per documentare il codice troppo rapidamente e superficialmente. Ciò, oltre a generare una riduzione di gran parti dei vantaggi derivanti da una buona documentazione, e quindi a ridurre la generale qualità del codice, può portare in casi limite a situazioni forvianti dovute alla presenza di commenti ingannevoli.

3.1.2 Mantenere i commenti aggiornati

Gli sviluppatori esperti redigono le prime versioni del codice in modo assolutamente professionale: il codice è chiaro, leggibile, ben documentato etc. Non è infrequente però il caso in cui, man mano che il codice subisce delle modifiche, la conseguente attività di aggiornamento dei commenti tenda ad essere trascurata. Ciò è particolarmente ricorrente a ridosso della data di consegna. Il risultato è la presenza di commenti non aggiornati e spesso contraddittori. Ciò è fonte di dannosa confusione che, in alcuni casi, finisce addirittura con il fuorviare e confondere il personale demandato alla manutenzione e quindi con il ridurre drasticamente il livello di manutenibilità del codice.

3.1.3 Scriveve commenti chiari e concisi

L'obiettivo dei commenti è di far comprendere l'intero codice prodotto sia allo stesso autore sia a persone estranee alla relativa implementazione. Pertanto è opportuno scrivere commenti concisi, semplici e chiari. Commenti troppo prolissi tendono a ridurre la leggibilità del codice. Tipicamente si preferisce scrivere i commenti utilizzando la terza persona singolare. Si supponga di disporre di una lista ordinata di codici di determinati prodotti e di basare il

metodo di ricerca su una determinata versione di un algoritmo di ricerca dicotomica (detta anche binaria). In questo caso nella documentazione è sufficiente riportare il nome dell'algoritmo ed eventualmente una brevissima spiegazione, evitando però di dilungarsi nell'esporre i dettagli di tale algoritmo per il quale esistono specifici trattati.

3.1.4 Evitare le decorazioni

Il codice prodotto deve essere commentato e particolare attenzione va rivolta alle porzioni di codice più complesse. I commenti, come visto in precedenza, devono essere professionali, chiari e concisi evitando inutili decorazioni e inappropriati umorismi. Questi, oltre a richiedere tempo che potrebbe essere investito in maniera migliore, tendono a diminuire la leggibilità del codice, in quanto ne diminuiscono pulizia e chiarezza, si prestano ad essere fraintesi e spesso possono irritare il lettore del codice impegnato a comprenderne il senso.

Listato 1 – Esempio di metodo printStackTrace *(classe* Throwable*) appositamente mal commentato.*

```
/**
 * -------------------------------------------------------------------------------------
 * Prints this throwable and its backtrace to the specified print stream.
 * -------------------------------------------------------------------------------------
 * @param s <code>PrintStream</code> to use for output
 * -------------------------------------------------------------------------------------
 */
public void printStackTrace(PrintStream s) {
  // ********************************************************************************
  // * It is necessary to synchronize the code to make it thread safe
  // ********************************************************************************
  synchronized (s) {
    s.println(this);

    // ********************************************************************************
    // * Prints all trace elements present in the stack
    // ********************************************************************************
    StackTraceElement[] trace = getOurStackTrace();
    for (int i=0; i < trace.length; i++)
    s.println("\tat " + trace[i]);

    // ********************************************************************************
    // * Prints the cause, if present
    // ********************************************************************************
    Throwable ourCause = getCause();
```

```
        if (ourCause != null)
        ourCause.printStackTraceAsCause(s, trace);
    }
}
```

3.1.5 Spiegare "che cosa" il codice esegue, "il perché" e non "il come"

L'illustrazione del "come" il codice risolva un compito, frequentemente, è una documentazione poco effettiva; infatti, le persone interessate a comprendere e a mantenere il codice sono, a loro volta, sviluppatori. Anche qualora non conoscano specifiche istruzioni o librerie, esistono molte fonti a cui attingere per la necessaria documentazione. Quello che invece è più difficile da comprendere è "che cosa" il programma intenda eseguire e "il perché". Pertanto, questi sono gli elementi sui quali è più opportuno investire il proprio tempo a disposizione.

Per esempio, si consideri il caso di un metodo la cui implementazione acceda sempre al primo elemento di un array riportante i valori di offerta e acquisto di un determinato strumento finanziario. Sebbene che cosa faccia un codice di questo tipo sia inequivocabile, potrebbe essere meno chiaro il perché, che, sempre nel caso ipotetico, potrebbe dipendere dal fatto che la prima posizione dell'array (indice = 0) sia riservata al prezzo più aggiornato.

3.1.6 Porre attenzione alle situazioni in cui sia difficile commentare il codice

Le situazioni in cui risulti difficile commentare efficacemente e semplicemente il codice prodotto dovrebbero essere oggetto di particolare attenzione. Infatti spesso situazioni del genere sono dovute a codice non ben scritto o paradossalmente non ben compreso.

3.1.7 Una buona documentazione inizia dal codice

È appena il caso di ricordare che una buona documentazione inizia dal codice; pertanto, è necessario investire nella leggibilità fin dalla fase di codifica.

3.2 Scrivere i commenti JavaDoc

JavaDoc è uno degli strumenti molto apprezzati inclusi nel JDK fin dalle sue prime versioni. In particolare, permette di organizzare razionalmente e produrre (di default) un insieme di pagine HTML contenenti la documentazione del codice prodotto. JavaDoc è sicuramente uno degli strumenti di maggiore successo del JDK tanto che anche altri linguaggi sono stati dotati di versioni simili di questa utility.

Comunque, nel redigere la documentazione JavaDoc è necessario porre attenzione a un insieme di regole che, se non seguite, potrebbero portare a risultati indesiderati.

3.2.1 Investire nella documentazione JavaDoc

Anche se questa regola può sembrare ormai acquisita, è sempre opportuno ricordare che una buona documentazione JavaDoc è caratteristica fondamentale di codici Java di buona

qualità. Questa documentazione fornisce il punto di partenza che i programmatori consultano ogniqualvolta hanno a che fare con del nuovo codice.

Inoltre, la maggior parte dei software IDE hanno servizi avanzati per assistere la produzione di commenti Javadoc. Quindi con minimo sforzo è possibile ottenere ottimi risultati.

3.2.2 Applicare la struttura dei *commenti doc*

I *commenti doc* devono essere specificati immediatamente prima dell'elemento (classe, interfaccia, metodo, attributo o particolare porzione di codice) di cui forniscono la documentazione. La struttura prevede una concisa e completa descrizione dell'elemento, eventualmente, seguita da un insieme di tag. La descrizione iniziale dovrebbe essere costituita da una prima riga che descriva, quanto più sinteticamente e completamente possibile, l'elemento al fine di fornirne un'immediata comprensione. Questa, tipicamente, è seguita da altre necessarie per fornire ulteriori informazioni. JavaDoc considera terminata la prima riga quando incontra un carattere di nuova riga o non appena incontra una sequenza di tipo: carattere punto e uno spazio oppure carattere punto e un tab. Pertanto bisogna porre attenzione ad eventuali contrazioni inserite nella prima riga che potrebbero, involontariamente, spezzare prematuramente il commento. Qualora un commento preveda più paragrafi, questi vanno separati per mezzo di una linea che contenga unicamente l'asterisco iniziale e il tag HTML <p> (cfr. quarta riga del commento riportato di seguito). Da tener presente che dalla versione 1.4 di JavaDoc, non è più necessario riportare il carattere asterisco iniziale nelle linee interne del commento. Per esempio si consideri la documentazione JavaDoc fornita con la classe System.

Listato 2 – Documentazione doc della classe java.lang.System.

```
/**
 * The <code>System</code> class contains several useful class fields
 * and methods. It cannot be instantiated.
 *<p>
 * Among the facilities provided by the <code>System</code> class
 * are standard input, standard output, and error output streams;
 * access to externally defined properties and environment
 * variables; a means of loading files and libraries; and a utility
 * method for quickly copying a portion of an array.
 *
 * @author  unascribed
 * @version 1.149, 06/02/04
 * @since   JDK1.0
 */
public final class System {
```

3.2.3 Scrivere i *commenti doc* utilizzando i tag HTML di formattazione

Il comportamento di default dell'utility JavaDoc, utilizzato nella stragrande maggioranza dei casi, consiste nell'inserire i commenti prelevati dal codice in una serie di pagine HTML

opportunamente organizzate. Quindi, a meno di voler ridefinire il comportamento di default di JavaDoc (a tal proposito è necessario utilizzare la relativa API), quando si redigono *commenti doc*, è necessario tenere a mente che questi verranno inseriti all'interno di una pagina HTML. Pertanto è opportuno porre attenzione all'utilizzo, nei commenti JavaDoc, dei caratteri considerati speciali per l'HTML. Per esempio, le parentesi angolari < > vanno sostituite, rispettivamente, dalle stringhe &lt e &gt. La stessa sorte spetta al carattere & che va sostituito con la stringa &amp, e così via. Inoltre, per assicurare che la formattazione utilizzata nello scrivere i vari commenti sia rispettata nelle pagine HTML generate, è opportuno ricorrere all'utilizzo di tag HTML quali <pre>, <b>, <i>, e così via.

Da tenere presente che, qualora si intenda scrivere il proprio doclet per la produzione di documentazione JavaDoc, questo si dovrà far carico di interpretare correttamente i tag HTML inclusi nella documentazione.

Per esempio, si consideri il seguente commento attinto dalla classe System relativo alle proprietà di sistema, riportato nel listato 3.

Listato 3 – Esempio di commento JavaDoc con tag HTML presente nella classe java.lang.System.

```
/**
   * System properties. The following properties are guaranteed to be defined:
   * <dl>
   * <dt>java.version         <dd>Java version number
   * <dt>java.vendor          <dd>Java vendor specific string
   * <dt>java.vendor.url      <dd>Java vendor URL
   * <dt>java.home            <dd>Java installation directory
   * <dt>java.class.version     <dd>Java class version number
   * <dt>java.class.path      <dd>Java classpath
   * <dt>os.name                <dd>Operating System Name
   * <dt>os.arch              <dd>Operating System Architecture
   * <dt>os.version             <dd>Operating System Version
   * <dt>file.separator         <dd>File separator ("/" on Unix)
   * <dt>path.separator       <dd>Path separator (":" on Unix)
   * <dt>line.separator       <dd>Line separator ("\n" on Unix)
   * <dt>user.name            <dd>User account name
   * <dt>user.home              <dd>User home directory
   * <dt>user.dir             <dd>User's current working directory
   * </dl>
   */
```

3.2.4 Utilizzare il tag <code>

Il tag <code> deve essere utilizzato per evidenziare elementi del codice. Pertanto va usato nei casi in cui il commento contenga parole chiavi java, nomi di package, nomi di classi, nomi di metodi, nomi di interfacce, attributi e argomenti e porzioni di codice.

Per esempio si consideri il JavaDoc relativo al metodo clearProperty dalla classe System.

Listato 4 – Commento doc del metodo clearProperty *della classe* java.lang.System.

```
/**
   * Removes the system property indicated by the specified key.
   * <p>
   * First, if a security manager exists, its
   * <code>SecurityManager.checkPermission</code> method
   * is called with a <code>PropertyPermission(key, "write")</code>
   * permission. This may result in a SecurityException being thrown.
   * If no exception is thrown, the specified property is removed.
   * <p>
   *
   * @param     key   the name of the system property to be removed.
   * @return    the previous string value of the system property,
   *               or <code>null</code> if there was no property with that key.
   *
   * @exception  SecurityException  if a security manager exists and its
   *             <code>checkPropertyAccess</code> method doesn't allow
   *             access to the specified system property.
   * @exception  NullPointerException if <code>key</code> is
   *             <code>null</code>.
   * @exception  IllegalArgumentException if <code>key</code> is empty.
   * @see    #getProperty
   * @see    #setProperty
   * @see    java.util.Properties
   * @see    java.lang.SecurityException
   * @see    java.lang.SecurityManager#checkPropertiesAccess()
   * @since 1.5
   */
  public static String clearProperty(String key) {
```

3.2.5 Evitare l'utilizzo di tag HTML di struttura

Come visto in precedenza è possibile e consigliato ricorrere all'utilizzo di alcuni tag HTML di formattazione per aumentare il livello di chiarezza dei *commenti doc*. Al contempo è fortemente consigliato che il ricorso a questi tag non includa elementi di struttura, come <HTML>, <HEAD>, <H1>, etc., al fine di non interferire con la preesistente struttura della pagina HTML utilizzata dall'applicazione JavaDoc; altrimenti si corre il forte rischio di generare pagine HTML assolutamente non leggibili (per esempio potrebbero comparire frame riportati nei posti più impensabili), o addirittura non riproducibili dai vari browser. Questa regola prevede l'eccezione della documentazione doc per i package, in quanto la preparazione dell'intera pagina è demandata al programmatore e quindi è assolutamente necessario ricorrere a tag HTML di struttura.

Non è infrequente il caso di commenti doc dei sorgenti JDK che utilizzano il tag <H4>. Comunque, utilizzare tale tag è un azzardo perché potrebbe creare problemi con evoluzioni future JavaDoc e/o con estensioni create dall'utente.

3.2.5 Fare attenzione all'inserimento di link

Nel produrre documentazione è spesso necessario inserire dei link (hyperlink) ad altri elementi. In questi casi è opportuno evitare il ricorso al tag HTML <A> e utilizzare al suo posto il tag {@Link}.

Il beneficio prodotto dall'utilizzo del tag link consiste nell'inserire un hyperlink all'interno della documentazione. Pertanto permette di saltare da una parte di documentazione all'altra. Poiché il pubblico dei fruitori di documentazione doc è costituito da personale esperto, è opportuno utilizzare con parsimonia questi link che richiedono tempo per essere inseriti e spesso rendono la documentazione meno chiara.

Un esempio di utilizzo del tag link è presente nel *commento doc* utilizzato per illustrare il metodo setProperties presente nella classe System.

Listato 5 – JavaDoc del metodo setProperties *della classe* java.lang.System.

```
/**
   * Sets the system properties to the <code>Properties</code>
   * argument.
   * <p>
   * First, if there is a security manager, its
   * <code>checkPropertiesAccess</code> method is called with no
   * arguments. This may result in a security exception.
   * <p>
   * The argument becomes the current set of system properties for use
   * by the {@link #getProperty(String)} method. If the argument is
   * <code>null</code>, then the current set of system properties is
   * forgotten.
   *
   * @param      props   the new system properties.
   * @exception  SecurityException  if a security manager exists and its
   *             <code>checkPropertiesAccess</code> method doesn't allow access
   *             to the system properties.
   * @see        #getProperties
   * @see        java.util.Properties
   * @see        java.lang.SecurityException
   * @see        java.lang.SecurityManager#checkPropertiesAccess()
   */
  public static void setProperties(Properties props) {
```

3.2.6 Ereditare i commenti JavaDoc

L'utility JavaDoc, in alcuni casi ben definiti, è in grado di duplicare ("ereditare") la documentazione doc dei metodi evitando al programmatore un'inutile riscrittura. Anche se ciò potrebbe essere evitato con un semplice copia e incolla, rimarrebbe il problema di dover mantenere aggiornate le ripetizione dello stesso *commento doc*.

JavaDoc è in grado di "ereditare" il JavaDoc dei metodi nelle seguenti situazioni:

1. un metodo di una classe esegue l'overriding del corrispondente metodo della superclasse (per esempio ogniqualvolta si scrive l'implementazione del metodo `toString()` di una classe);

2. un metodo di un'interfaccia esegue l'overriding del corrispondente metodo della superinterfaccia;

3. quando un metodo di una classe implementa un metodo di un'interfaccia.

3.2.7 Scrivere i commenti JavaDoc per tutti gli elementi

Il *commento doc* va scritto per tutti gli elementi del proprio codice. In particolare è importante documentare correttamente classi, interfacce, metodi e attributi. La relativa descrizione è riportata nelle sezioni di pertinenza illustrate di seguito.

3.3 Valutare la possibilità di inserire la documentazione al livello di package

Documentare i package è un'attività non strettamente obbligatoria che tuttavia fornisce importanti informazioni circa le classi e le interfacce contenute. Questa documentazione è particolarmente utile sia in tutti quei casi in cui il criterio di aggregazione di classi e interfacce all'interno di un medesimo package non sia particolarmente intuitivo, e sia qualora sia necessario includere informazioni relative all'intero package. Pertanto, questa documentazione offre la possibilità di concentrare informazioni comuni a diverse classi e interfacce che, altrimenti, dovrebbero essere ripetute nei vari elementi creando problemi di manutenibilità.

Un altro elemento da tenere in considerazione è relativo al fatto che, alla presenza di nuove librerie e nuovo codice da apprendere, tipicamente, gli sviluppatori avviano lo studio proprio a partire da queste informazioni poiché permettono di acquisire un'iniziale cognizione del ruolo e delle responsabilità dell'intero package, delle classi contenute, etc.

3.2.1 Utilizzare il formato standard di documentazione dei package

Con la versione 1.2 dell'utility JavaDoc è possibile inserire documentazione a livello di package che poi JavaDoc stesso è in grado di includere nella documentazione del codice generata. Queste informazioni devono essere incluse in un file HTML di nome fisso `package.html` inserito nell'apposito package insieme a classi e interfacce contenute. Il comportamento di JavaDoc

consiste nel prelevare tutte le informazioni contenute tra i tag <body> e </body>. I tag JavaDoc utilizzabili in questa documentazione sono @see, @link, @deprecated, @since.

Il template suggerito dalla Sun prevede le seguenti sezioni:

1. l'immancabile copyright dell'azienda;
2. commenti generali (tipicamente riportati subito dopo il tag HTML body), dove inserire una breve descrizione del package e dei vari servizi offerti;
3. documentazione specifica del package, tipicamente composta da:
   a. specifiche relative all'intero package; per esempio nel pacakge AWT sono riportate specifiche relative all'interazione del framework con i vari sistemi operativi;
   b. direttive per ulteriori tool di manipolazione del testo, come per esempio FrameMaker;
   c. riferimenti specifici;
4. riferimenti ad altra documentazione esterna, come articoli, libri, etc.

Di seguito è riportato il listatino HTML proposto dalla Sun come template per la documentazione dei package Java.

Listato 6 – Esempio di modello utilizzabile per includere la documentazione di package.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
<head>
<!--

@(#)package.html  1.60 98/01/27

                    Copyright (c) <anno copyright> <nome azienda>
                          All Rights Reserved

  This software is the confidential and proprietary information of <nome azienda>
  ("Confidential Information").  You shall not disclose such Confidential Information
  and shall use it only in accordance with the terms of the license agreement
  you entered into with <nome azienda>.

-->
```

```
</head>
<body bgcolor="white">

##### THIS IS THE TEMPLATE FOR THE PACKAGE DOC COMMENTS. #####
##### TYPE YOUR PACKAGE COMMENTS HERE.  BEGIN WITH A     #####
##### ONE-SENTENCE SUMMARY STARTING WITH A VERB LIKE:    #####
Provides for....


<h2>Package Specification</h2>

##### FILL IN ANY SPECS NEEDED BY JAVA COMPATIBILITY KIT #####
<ul>
  <li><a href="">##### REFER TO ANY FRAMEMAKER SPECIFICATION HERE #####</a>
</ul>

<h2>Related Documentation</h2>

For overviews, tutorials, examples, guides, and tool documentation, please see:
<ul>
  <li><a href="">##### REFER TO NON-SPEC DOCUMENTATION HERE #####</a>
</ul>

<!-- Put @see and @since tags down here. -->

</body>
</html>
```

Il seguente listato riporta un esempio di utilizzo della documentazione doc al livello di package.

Listato 7 – Frammento del JavaDoc relativo al package java.util.logging.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
<head>
<!--
@(#)package.html     1.7 04/06/17

Copyright 2004 Sun Microsystems, Inc. All rights reserved.
SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
-->
</head>
<body bgcolor="white">
```

```
<P>
Provides the classes and interfaces of
the Java<SUP><FONT SIZE="-2">TM</FONT></SUP> 2
platform's core logging facilities.
The central goal of the logging APIs is to support maintaining and servicing
software at customer sites.

<P>
There are four main target uses of the logs:
</P>

<OL>
  <LI> <I>Problem diagnosis by end users and system administrators</I>.
    This consists of simple logging of common problems that can be fixed
    or tracked locally, such as running out of resources, security failures,
    and simple configuration errors.

  <LI> <I>Problem diagnosis by field service engineers</I>. The logging information
    used by field service engineers may be considerably more complex and
    verbose than that required by system administrators.  Typically such information
    will require extra logging within particular subsystems.
</OL>
</P>
The key elements of this package include:
<UL>
  <LI> <I>Logger</I>: The main entity on which applications make
       logging calls. A Logger object is used to log messages
       for a specific system or application
       component.
  <LI> <I>LogRecord</I>: Used to pass logging requests between the logging
       framework and individual log handlers.
  <LI> <I>Handler</I>: Exports LogRecord objects to a variety of destinations
       including memory, output streams, consoles, files, and sockets.
       A variety of Handler subclasses exist for this purpose. Additional Handlers
       may be developed by third parties and delivered on top of the core platform.
  <LI> <I>Level</I>: Defines a set of standard logging levels that can be used
       to control logging output. Programs can be configured to output logging
       for some levels while ignoring output for others.
  <LI> <I>Filter</I>: Provides fine-grained control over what gets logged,
       beyond the control provided by log levels. The logging APIs support a general-purpose
       filter mechanism that allows application code to attach arbitrary filters to
       control logging output.
```

```
  <LI> <I>Formatter</I>: Provides support for formatting LogRecord objects. This
       package includes two formatters, SimpleFormatter and
       XMLFormatter, for formatting log records in plain text
       or XML respectively. As with Handlers, additional Formatters
       may be developed by third parties.
</UL>
<P>
The Logging APIs offer both static and dynamic configuration control.
Static control enables field service staff to set up a particular configuration and then re-launch the
application with the new logging settings. Dynamic control allows for updates to the
logging configuration within a currently running program. The APIs also allow for logging to be
enabled or disabled for different functional areas of the system. For example,
a field service engineer might be interested in tracing all AWT events, but might have no interest in
socket events or memory management.
</P>

<h2>Null Pointers</h2>
<p>
In general, unless otherwise noted in the javadoc, methods and
constructors will throw NullPointerException if passed a null argument.
The one broad exception to this rule is that the logging convenience
methods in the Logger class (the config, entering, exiting, fine, finer, finest,
log, logp, logrb, severe, throwing, and warning methods)
will accept null values
for all arguments except for the initial Level argument (if any).
<p>
<H2>Related Documentation</H2>
<P>
For an overview of control flow,
please refer to the
<a href="../../../../guide/logging/overview.html">
Java Logging Overview</a>.
</P>

<!-- Put @see and @since tags down here. -->

@since 1.4


</body>
</html>
```

3.4 Commentare correttamente le classi

La naturale tendenza del disegno OO consiste nel generare una serie di classi opportunamente relazionate tra loro. Commentare correttamente le classi permette di capirne il ruolo, le responsabilità e di inserirle rapidamente nel relativo contesto. Da tener presente che non sempre lo studio di un'API richiede di analizzare tutte le classi e, anche qualora ciò sia necessario, la documentazione a livello di classe aiuta a concentrarsi, in ogni momento, su un numero limitato di aspetti. Per esempio, mentre si analizza una classe, è possibile, temporaneamente, studiare esclusivamente la descrizione generale delle altre relazionate, senza dover necessariamente scendere, fin da subito, nei relativi dettagli.

3.4.1 Includere sempre la dichiarazione del copyright

Tutti i codici sorgenti dovrebbero essere dotati di una sezione iniziale in cui sono specificati i termini di copyright. Questa parte deve essere riportata a partire dalla prima riga.

L'esempio classico è riportato nel listato 8, ove le stringhe *<anno copyright>* e *<nome azienda>*, come chiaramente indicato dai relativi nomi, vanno sostituiti, rispettivamente, con l'anno del copyright ed il nome dell'azienda. Per esempio: 2006 e MokaByte Srl.

Listato 8 – Esempio di indicazione di copyright.

```
/*
*                    Copyright (c) <anno copyright> <nome azienda>
*                          All Rights Reserved
*
* This software is the confidential and proprietary information of <nome azienda>
* ("Confidential Information").  You shall not disclose such Confidential Information
* and shall use it only in accordance with the terms of the license agreement
* you entered into with <nome azienda>.
*/
```

3.4.2 Includere sempre un'intestazione per classi e interfacce

Ogni classe e interfaccia, secondo anche i dettami dei commenti JavaDoc ([SJAVCC]), deve essere introdotta da un'opportuna intestazione (*header* in Inglese) che riporti una breve descrizione della classe, i suoi ruoli, le principali responsabilità, l'indicazione che specifichi se si tratti o meno di una classe persistente, la versione del JDK di riferimento, la lista degli autori, la versione e la data dell'ultimo aggiornamento ed eventuali riferimenti ad ulteriore documentazione, classi ed interfacce relazionate alla presente.

Per classi il cui utilizzo possa non risultare immediato, è inoltre consigliato riportare alcuni esempi di utilizzo in termini di codice. Un'altra buona norma prevede di riportare una breve descrizione di eventuali problemi, limitazioni, etc. della classe. Infine, è da notare che dalla versione Tiger (JDK 1.2.5), i linguaggio Java è stato (finalmente) dotato del concetto di classi template, denominato *generics*. Questo fa sì che si possano avere classi parametriche. Pertan-

to, in questo caso, l'header deve riportare anche la descrizione dei parametri della classe (@param).

Un esempio di tale intestazione è mostrata nel listato 9.

Listato 9 – Esempio di un template *per l'intestazione di classi ed interfacce.*

```
/**
 * <inserire qui una breve descrizione, possibilmente di una riga >
 * <inserire ulteriori informazioni >
 * <p>
 * <b>Responsibilities:</b><ul>
 * <li><inserire qui l'elenco delle principali responsabilità della classe>
 * </ul>
 * <p><code>
 * <esempi di codice>
 * </code>
 *
 * <b>Persistent:</b>Yes/No<br>
 *
 * @since   JDK<x.x.x>
 * @author  <nome degli autori>
 * @version <x.xx.xxx> – <data dell'ultima modifica>
 * @param <parametro> – <descrizione del parametro>
 *
 * @see <altre classi/interfacce relazionate>
 */
```

Listato 10 – Frammento dell'header della classe java.util.concurrent.Future.

```
/**
 * A <tt>Future</tt> represents the result of an asynchronous
 * computation.  Methods are provided to check if the computation is
 * complete, to wait for its completion, and to retrieve the result of
 * the computation.  The result can only be retrieved using method
 * <tt>get</tt> when the computation has completed, blocking if
 * necessary until it is ready.  Cancellation is performed by the
 * <tt>cancel</tt> method.  Additional methods are provided to
 * determine if the task completed normally or was cancelled. Once a
 * computation has completed, the computation cannot be cancelled.
 * If you would like to use a <tt>Future</tt> for the sake
 * of cancellability but not provide a usable result, you can
 * declare types of the form <tt>Future&lt;?&gt;</tt> and
 * return <tt>null</tt> as a result of the underlying task.
 *
```

```
 * <p>
 * <b>Sample Usage</b> (Note that the following classes are all
 * made-up.) <p>
 * <pre>
 * interface ArchiveSearcher { String search(String target); }
 * class App {
 *ExecutorService executor = ...
 *ArchiveSearcher searcher = ...
 *void showSearch(final String target) throws InterruptedException {
 *    Future&lt;String&gt; future = executor.submit(new Callable&lt;String&gt;() {
 *        public String call() { return searcher.search(target); }
 *    });
 *    displayOtherThings(); // do other things while searching
 *    try {
 *        displayText(future.get()); // use future
 *    } catch (ExecutionException ex) { cleanup(); return; }
 *    }
 * }
 * </pre>
 * @see FutureTask
 * @see Executor
 * @since 1.5
 * @author Doug Lea
 * @param <V> The result type returned by this Future's <tt>get</tt> method
 */
public interface Future<V> {
```

3.4.3 Non inserire la "history"

Alcuni testi di informatica suggeriscono di aggiungere nell'intestazione di classi e interfacce una breve storia inerente i vari processi di modifica subiti. Questa dovrebbe prevedere una lista di linee aventi un formato del tipo: data, autore, modifica. Per esempio:

```
10/Oct/2005 - L.V.T. - Introdotto metodo per il riordino automatico degli elementi della lista.
```

Sebbene si tratti di informazioni indubbiamente utili e interessanti, si ritiene che il codice non sia il luogo più opportuno ove includerle. In particolare, queste informazioni dovrebbero essere di pertinenza di sistemi di controllo dei sorgenti (*source control*) quali: CVS, IBM Rational Clear Case, Ms Source Safe, etc. La presenza di queste informazioni, dopo qualche iterazione del sistema e conseguenti aggiornamenti del codice, tende a diventare notevole e a rendere il codice più pesante e meno chiaro. Inoltre, superate le cinque, sei righe, queste informazioni, insieme alle altre presenti nell'intestazione delle classi ed interfacce, tendono ad essere sistematicamente tralasciate e quindi a divenire meri dati.

3.4.4 Commentare chiaramente problemi e limitazioni del codice

Alcune volte accade che l'implementazione di una classe o interfaccia presenti problemi o limitazioni la cui risoluzione non sia possibile o in assoluto, magari per dipendenze dall'hardware o da software fornito da terze parti, o semplicemente in quel momento per una serie di motivazioni. Alcuni esempi sono dati dalla presenza di una grande quantità di codice dipendente da quello da modificare che quindi subirebbe serie ripercussioni da eventuali modifiche, oppure dalla presenza di dipendenze da altro codice, da situazioni in cui la soluzione dei problemi e/o limitazioni non sia d'interesse per la versione attuale del sistema, magari per mancanza di tempo e/ budget.

Comunque sia, in questi casi è necessario documentare dettagliatamente questi problemi e limitazioni. Il punto migliore dove includere queste informazioni è nell'intestazione della classe o interfaccia (cfr. punto precedente).

Documentare i problemi di una classe è particolarmente importante per il riutilizzo della stessa, in quanto fornisce agli sviluppatori una serie di informazioni utili per prendere decisioni, qualora sia possibile o meno procedere con il riutilizzo del codice. Inoltre, è molto utile per la produzione di versioni successive del sistema e, in taluni casi, anche per la fase di test.

Listato 11 – Parte della documentazione della classe java.sql.Timestamp.

```
 * <P><B>Note:</B> This type is a composite of a <code>java.util.Date</code> and a
 * separate nanoseconds value. Only integral seconds are stored in the
 * <code>java.util.Date</code> component. The fractional seconds - the nanos - are
 * separate.  The <code>Timestamp.equals(Object)</code> method never returns
 * <code>true</code> when passed a value of type <code>java.util.Date</code>
 * because the nanos component of a date is unknown.
 * As a result, the <code>Timestamp.equals(Object)</code>
 * method is not symmetric with respect to the
 * <code>java.util.Date.equals(Object)</code>
 * method.  Also, the <code>hashcode</code> method uses the underlying
 * <code>java.util.Date</code>
 * implementation and therefore does not include nanos in its computation.
 * <P>
 * Due to the differences between the <code>Timestamp</code> class
 * and the <code>java.util.Date</code>
 * class mentioned above, it is recommended that code not view
 * <code>Timestamp</code> values generically as an instance of
 * <code>java.util.Date</code>.  The
 * inheritance relationship between <code>Timestamp</code>
 * and <code>java.util.Date</code> really
 * denotes implementation inheritance, and not type inheritance.
```

3.4.5 Commentare l'ordine di esecuzione

Non è infrequente il caso in cui l'implementazione di un metodo o i servizi offerti da una determinata classe debbano essere eseguiti rispettando un ben definito ordine non immediatamente comprensibile. In questi casi è fortemente consigliato documentare attentamente tale ordine di esecuzione. Nel caso ci si riferisca all'implementazione di un metodo, queste informazioni sono necessarie solo al personale addetto alla relativa manutenzione e quindi possono essere inserite nell'intestazione del metodo e/o all'interno del metodo stesso. Qualora invece si voglia documentare l'ordine di esecuzione dei metodi di una classe, (consistentemente con quando riportato al punto precedente) è necessario riportare tali informazioni nell'intestazione della classe ed, eventualmente, inserire un rimando nell'intestazione dei vari metodi.

3.4.6 Commentare eventuali scelte/soluzioni non chiare

Nell'implementare una classe, un metodo o addirittura nella definizione della firma dei metodi che costituiscono un'interfaccia, non è infrequente il caso in cui si ricorra a soluzioni che inizialmente potrebbero sembrare poco chiare o addirittura errate ma che, invece, sono il frutto di un attento studio e di una lunga valutazione. Tali scelte, per esempio, potrebbero essere relative all'utilizzo di specifiche strutture dati invece di altre, alla concorrenza, alla gestione delle eccezioni, alla visibilità di taluni metodi, e così via. In tutti questi casi è opportuno documentare chiaramente le soluzioni adottate incluse le relative motivazioni. Questa documentazione è essenziale per futura memoria, per non rischiare ingiustificate riscritture del codice, per evitarne un cattivo uso, per non ripetere dibattiti avvenuti in precedenza, e così via.

3.4.7 Commentare eventuali effetti collaterali a livello di sistema

Spesso l'esecuzione dei metodi di una classe genera effetti collaterali, non ovvi, all'interno del sistema. In questi casi è opportuno documentare chiaramente tali effetti nell'intestazione della classe.

3.4.8 Documentare tutti i metodi

Una buona tecnica di documentazione del codice, rinforzata dalla convenzione JavaDoc ([SJAVCC]), consiste nell'introdurre tutti i metodi con una ben definita sezione di commento che includa una breve descrizione del metodo (una riga, seguita da un commento più lungo, come illustrato nella sezione JavaDoc), un'eventuale descrizione dei parametri formali, del valore di ritorno se presente, delle eventuali eccezioni e, qualora sia il caso, l'indicazione della deprecazione, come indicato dal listato 12.

Listato 12 – Modello per la documentazione dei metodi.

```
/**
 * <Descrizione del metodo sintetica>
 * <p>
 * <Ulteriori informazioni relative al metodo sintetica>
```

```
*
* @param          <nome del parametro>  <descrizione del parametro>
* @param          <nome del parametro>  <descrizione del parametro>
* @return         <valore di ritorno se presente>
* @see            <eventuali rimandi a classi/interface contenenti informazioni utili>
* @throws         <classe eccezione> <spiegazione delle cause che la generano>
* @throws         <classe eccezione> <spiegazione delle cause che la generano>
* @deprecated     <descrizione della deprecaizone>
*/
```

Listato 13 – Esempio di commento doc relativo al metodo log *della classe* java.util.jogging.Logger.

```
/**
  * Log a LogRecord.
  * <p>
  * All the other logging methods in this class call through
  * this method to actually perform any logging.  Subclasses can
  * override this single method to capture all log activity.
  *
  * @param record the LogRecord to be published
  */
 public void log(LogRecord record) {
```

Come al solito, nella descrizione del metodo è importante descrivere cosa il metodo fa e non come lo da. Le descrizioni dovrebbero essere indipendenti dall'implementazione. Inoltre, qualora non sia immediato il perché il codice esegua determinati compiti è consigliabile aggiungere anche una breve descrizione di ciò. Queste informazioni aiutano a inserire il metodo nel relativo contesto e quindi ne semplificano la leggibilità e riutilizzabilità.

3.4.9 Documentare gli attributi

Gli attributi, specie quelli di classe tendono a sfuggire all'attenzione dei programmatori che, a volte, ne trascurano la documentazione. Invece, anche questa assume un ruolo di primaria importanza specie quando il relativo utilizzo non è immediato, e per la corretta comprensione di algoritmi complessi. Come di consueto, è importante riportare una documentazione chiara e concisa che spieghi l'utilizzo dell'attributo. In casi particolarmente complessi o poco intuitivi, è consigliabile anche mostrare anche alcuni esempi di utilizzo.

Per gli attributi di classe (il cui campo d'azione è l'intera classe) è consigliabile ricorrere a una documentazione in stile JavaDoc, specie per attributi la cui visibilità non è privata. Per gli attributi dei metodi (comunemente denominati variabili) è invece sufficiente un commento in linea.

Al fine di mantenere il codice pulito e chiaro è consigliabile riportare una dichiarazione per riga. Di seguito è mostrata la documentazione della variabile boolean della classe java.lang.Boolean e mostrato un attributo dotato di documentazione JavaDoc.

Listato 14 – Esempio di documentazione JavaDoc di un attributo di classe.

```
/**
 * The value of the Boolean.
 * @serial
 */
private final boolean value;
```

Il seguente è invece un commento in linea relativo all'attributo c presente nell'implementazione del metodo put(E o) della classe java.util.concurrent.LinkedBlockingQueue.

Listato 15 – Esempio di documentazione di una variabile di un metodo.

```
// Note: convention in all put/take/etc is to preset
// local var holding count  negative to indicate failure unless set.
int c = -1;
```

3.5 Commentare attentamente l'implementazione

Il linguaggio di programmazione Java prevede ben tre differenti stili di documentazione del codice (anche questa ricchezza di stili è l'ennesima dimostrazione dell'importanza attribuita alla documentazione del software), che sono:

- commenti JavaDoc: questi, come visto in precedenza, sono riconoscibili in quanto racchiusi tra i delimitatori /** e */ (/** commento */);
- commenti in stile linguaggio C caratterizzati dal formato /* commento */;
- commento di singola linea che consiste nel considerare come commento tutto il testo che segue due caratteri barretta obliqua: // commento.

Ogni stile presenta specifiche caratteristiche che ne rendono opportuno l'utilizzo in determinati ambiti a discapito di altri. Per esempio i commenti in JavaDoc sono quelli prelevati dall'omonima utility e inseriti nella documentazione generata automaticamente da tale tool. Il relativo utilizzo è pertanto consigliato come introduzione a dichiarazioni di classe, interfacce, metodi e attributi, mentre più raro è il relativo utilizzo all'interno del codice.

I commenti stile C sono spesso utilizzati in tutti quei casi in cui il testo di commento richieda diverse linee. Inoltre risulta particolarmente utile durante la fase di debbugging qualora si renda necessario isolare porzioni di codice per individuare la parte di codice errata.

I commenti di singola linea sono tipicamente utilizzati per commentare variabili locali a metodi e opportune porzioni di codice. Un'interessante convenzione consiste nell'allineare questi commenti al margine destro. Tuttavia, quantunque tale convenzione sia in grado di

produrre un effetto gradevole in alcuni casi specifici, in diverse situazioni, l'ottenimento dell'effetto voluto richiede un notevole investimento di tempo non sempre giustificato.

3.5.1 Documentare efficacemente il codice

I commenti presenti all'interno dei metodi sono tra i più importanti in quanto documentano l'implementazione vera e propria. Per questo motivo è necessario porre particolare attenzione alle consuete regole: scrivere commenti concisi e, allo stesso tempo, completi, porsi sempre nei panni di uno sviluppatore completamente all'oscuro del codice e delle relative decisioni e ricordarsi di indicare le motivazioni alla base del codice (il famoso "perché").

Per esempio si consideri l'implementazione del metodo writeObject (utilizzato per serializzare lo stato dell'oggetto) della classe java.util.ArrayList.

Listato 16 – Implementazione del metodo writeObject della classe java.util.ArrayList.

```
/**
 * Saves the state of the <tt>ArrayList</tt> instance to a stream (that
 * is, serialize it).
 *
 * @param      s object output stream used to serialize the object
 * @throws     IOException IO problems occurred during the serialization
 * @serialData The length of the array backing the <tt>ArrayList</tt>
 *             instance is emitted (int), followed by all of its elements
 *             (each an <tt>Object</tt>) in the proper order.
*/
private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException {

  // Write out element count, and any hidden stuff
  s.defaultWriteObject();

  // Write out array length
  s.writeInt(elementData.length);

  // Write out all elements in the proper order.
  for (int i=0; i<size; i++)
    s.writeObject(elementData[i]);
}
```

3.5.2 Ricordarsi di rimuovere parti di codice non più utilizzate

Spesso, durante la messa a punto del codice prodotto o a seguito di operazioni di manutenzione dello stesso, si procede con l'isolare determinate porzioni di codice escludendone altre racchiudendole in appositi commenti. Sebbene questa sia una prassi conveniente, può

accadere di lasciare commentate delle porzioni di codice perché erronee o non più necessarie. In questi casi è opportuno rivedere il codice e rimuovere tali parti prima di consolidare il codice. Infatti, porzioni di sorgente racchiuse in un commento (e non parte integrante della documentazione), e quindi non più utilizzate, finiscono per confondere la lettura del codice stesso e tendono a creare amletici interrogativi nel personale addetto alla manutenzione.

Se proprio, per qualche importante motivo (per esempio un servizio da utilizzare in una versione futura) non si voglia perdere tale porzione di codice, allora è importante riportare una descrizione che spieghi sia il motivo per cui il codice è stato commentato, sia la data e l'autore della trasformazione in commento.

3.5.3 Documentare efficacemente cicli annidati

Un codice complesso che includa diversi cicli annidati, ognuno caratterizzato da specifiche comparazioni, raramente risulta leggibile. Pertanto, in circostanze del genere dovrebbe essere naturale valutare la possibilità di aumentarne la leggibilità, magari delegando porzioni di codice a opportuni metodi privati. Qualora ciò non sia possibile, allora è fondamentale commentare attentamente il codice, evidenziando opportunamente i vari cicli. Spesso non è sufficiente delegare questa responsabilità alla sola indentazione del codice. In presenza di diversi cicli annidati, quindi è consigliabile commentare attentamente la chiusura di ciascun ciclo con commenti del tipo } // end of if, } // end of while, e così via.

L'esempio del listato 17 è tratto dall'implementazione del metodo indexOf della classe AbstractList.

Listato 17 – Esempio del codice di implementazione del metodo indexOf *della classe* java.util.AbstractList*. Si notino i commenti aggiunti dall'autore..*

```
public int indexOf(Object o) {
  ListIterator<E> e = listIterator();
  if (o==null) {
    while (e.hasNext()) {
      if (e.next() == null) {
        return e.previousIndex();
      } // end of if
    } // end of while
  } else {
    while (e.hasNext()) {
      if ( o.equals(e.next()) ) {
        return e.previousIndex();
       } // end of if
    } // end of while
  } // end of else
  return -1;
}
```

3.5.4 Documentare eventuali effetti collaterali

Al fine di produrre un buon livello di documentazione è necessario documentare chiaramente eventuali effetti collaterali, non ovvi, generati dall'esecuzione dei metodi. Come di consueto non è necessario documentare casi lampanti come, per esempio, i metodi modificatori (setXXX) disegnati specificatamente per generare un effetto collaterale (modificare il valore di una o più attributi).

Si consideri il metodo riportato nel listato 18. Questo si occupa si reimpostare la posizione del Buffer. Come riportato nella documentazione doc, questa operazione può avere effetto sull'attributo mark. Si tratta appunto di un necessario effetto collaterale.

Listato 18 – Implementazione del metodo position *della classe* java.nio.Buffer.

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

  position = newPosition;

  if (mark > position)
    mark = -1;

  return this;
}
```

3.5.5 Documentare eventuali limitazioni dell'implementazione di un metodo

Analogamente a quanto riportato a livello di classe (cfr. 3.4.4), non è infrequente il caso in cui l'implementazione di un metodo presenti qualche problema e/o limitazione che, magari per questione di tempo o per la presenza di dipendenze da altre parti di codice, non possano essere immediatamente sistemate. In questi casi è opportuno documentare chiaramente tali problemi e/o limitazioni. Ciò è fondamentale per semplificare la manutenzione e ricusabilità di tale codice.

Listato 19 – Commento del metodo nanoTime *della classe* java.lang.System.

```
/**
 * Returns the current value of the most precise available system
 * timer, in nanoseconds.
 *
 * <p>This method can only be used to measure elapsed time and is
 * not related to any other notion of system or wall-clock time.
 * The value returned represents nanoseconds since some fixed but
 * arbitrary time (perhaps in the future, so values may be
 * negative).  This method provides nanosecond precision, but not
 * necessarily nanosecond accuracy. No guarantees are made about
 * how frequently values change. Differences in successive calls
 * that span greater than approximately 292 years (2<sup>63</sup>
 * nanoseconds) will not accurately compute elapsed time due to
 * numerical overflow.
 *
 * <p> For example, to measure how long some code takes to execute:
 * <pre>
 *long startTime = System.nanoTime();
 *// ... the code being measured ...
 *long estimatedTime = System.nanoTime() - startTime;
 * </pre>
 *
 * @return The current value of the system timer, in nanoseconds.
 * @since 1.5
 */
public static native long nanoTime();
```