

Appunti Java

1 - Il Linguaggio

A cura di R. Gimelli, G. Necordi, M. Campodonico

V 1.0 - 1/2003

Copyright (c) 2003 Jtech.Srl - www.jtech.it

Sommario

1. Abstract.....	5
2. Condizioni d'uso del documento.....	5
3. Obiettivi del linguaggio.....	5
4. JVM Java Virtual Machine.....	5
5. Identificatori.....	6
6. Tipi dati elementari.....	7
7. Operatori.....	8
8. Dichiarazioni.....	9
8.1 Dichiarazione di classe.....	9
8.2 Dichiarazione di interfaccia.....	9
8.3 Dichiarazione di metodi.....	9
8.4 Dichiarazione del main.....	9
8.5 Dichiarazione del costruttore.....	9
8.6 Dichiarazione ed inizializzazione di variabili.....	10
9. Convenzioni nomenclatura.....	10
10. Visibilità di classi e metodi.....	11
10.1 Visibilità di classe.....	11
10.2 Visibilità di membri (attributi o metodi).....	11
11. Controllo di flusso.....	11
11.1 statement di scelta: if-then-else.....	11
11.2 Statement di scelta: switch-case-default.....	12
11.3 Statement di iterazione: for.....	13
11.4 statement di iterazione: while.....	13
12. Gestione delle Exception.....	14
13. La Garbage Collection.....	15
13.1 Algoritmi di garbage collection.....	15
13.2 Finalize.....	15
13.3 Referenze.....	15
14. Encapsulation in OOD.....	17
14.1 Benefici dell'OOD.....	17
14.2 Concetti base sulle relazioni tra oggetti.....	17
15. Ridefinizioni dei metodi.....	17
15.1 Metodi overridden.....	17
15.2 Metodi Overloaded.....	18
15.3 Costruttori overloaded e overridden.....	19
16. Inner classes.....	19
16.1 Static inner classes.....	19
16.2 Member inner classe.....	19
16.3 Local inner classes.....	19
16.4 Anonymous inner classes.....	19
17. Collections.....	20
17.1 Definizione di Collection.....	20
17.2 Definizione di Set.....	21
17.3 Definizione di List.....	21
17.4 Definizione di Map.....	22
17.5 Sintesi su Collection.....	23

17.6Wrapper Implementation.....	23
18.Threads.....	23
18.1Creazione.....	23
18.2Avvio.....	23
18.3Sleep e interrupt.....	24
18.4Concorrenza.....	24
18.5Scheduling.....	24
18.6ThreadGroups, security e setDaemon().....	24
19.Bibliografia e riferimenti.....	24
20.Marchi.....	25

1. Abstract

Il documento si prefigge l'obiettivo di descrivere in maniera sintetica i fondamenti del linguaggio JAVA. Questo documento non è né un reference book né un tutorial sul linguaggio JAVA: è nato con l'intendimento di fornire ai candidati all'esame di certificazione SUN lo strumento per un rapido ripasso.

Per la preparazione all'esame diverso materiale - libri e documentazione elettronica - è disponibile. SUN rimane comunque il punto di riferimento, sia per quanto riguarda il training che la documentazione (www.java.sun.com/docs/).

2. Condizioni d'uso del documento

Il documento è di proprietà degli autori che concedono il diritto d'uso di copia, diffusione e pubblicazione a condizione che venga riconosciuta la proprietà intellettuale dello stesso.

Il documento viene fornito senza alcuna garanzia implicita ed esplicita e nessuna responsabilità può essere ricondotta agli autori per eventuali danni - di qualunque genere e natura - che dovessero da esso derivare.

3. Obiettivi del linguaggio

- x Linguaggio di programmazione ad oggetti, multi threading, multiplatforma, erede del C
- x Sicuro, sia per quanto riguarda possibili bug dovuti a debolezza (o flessibilità?) del linguaggio di programmazione (puntatori, memoria), sia per quanto riguarda esecuzione esente da codice dannoso

4. JVM Java Virtual Machine

Java -> multiplatforma = linguaggio semi interpretato. Per ogni piattaforma software viene costruita una macchina virtuale software che sia in grado di eseguire il codice 'compilato' Java (file .class).

In realtà oggi Java non è più semi interpretato ma compilato dinamicamente (tecnologia jit - just in time compiler - o hotspot - compilazione dinamica jit con ottimizzazione dei processi).

Il programma Java può essere anche eseguito all'interno di un browser da una JVM integrata se presente; in questo caso prende il nome di *applet*.

Alcuni particolari programmi Java possono essere eseguiti all'interno di un *application server*, cioè un server che fornisce il supporto Java ad un *web server*; questi programmi prendono il nome di *servlet* o *pagine JSP*.

In breve
Da sorgente a codice – Compilazione - : <code>javac <nome file.java></code> . Genera file <code>.class</code> .
Esecuzione applicazioni – JVM per applicazioni -: <code>java <nome classe></code> .
CLASSPATH: variabile di environment che indica alla JVM dove trovare i file <code>.class</code>
File jar e Zip. File che contengono eseguibili (file <code>.class</code>) e risorse (immagini, testi, ecc.).

Durante il runtime, l' invocazione della JVM con il comando `java <nome classe>` provoca:

Fase	Descrizione
ClassLoader	Il modulo si occupa del caricamento delle classi necessarie all' esecuzione del programma, separando nomi locali da nomi di rete e caricando prima le classi predefinite.
Byte code verifier	Ha la responsabilità di verificare il formato delle classi, le verifiche sulla security degli accessi richiesti, l' utilizzo dello stack, i tipi dei parametri, conversioni ed accessi ai campi delle classi.
Interprete o generatore JIT	Il codice può essere interpretato dalla JVM o compilato da un generatore just in time (jit) o da, 1.2, compilato con tecniche di buffering e caching (motore hot spot).

N.B.:

Un file `.java` può contenere la definizione di un solo file class di tipo pubblico.

5. Identificatori

Regole per il compilatore:

- x Un identificatore può iniziare con una lettera, `_` o `$` e può contenere, lettere, cifre, numeri, `_` e `$`.
- x Non deve essere una parola riservata come `if`, `switch`, ecc.

Esistono inoltre delle **convenzioni di nomenclatura** per definire il nome degli identificatori (v. Convenzioni di nomenclatura).

6. Tipi dati elementari

Rappresentano gli elementi atomici immutabili delle strutture dati.

Non sono oggetti ma esistono classi denominate ' wrapperclasses' che consentono la creazione di omologhi oggetti.

Tipo	Dimensione	Valori	Letterali
byte	8 bit	-128 ÷ +127	es. 12, 014 , 0x0C
short	16 bit	-32768 ÷ +32767	es. 300, 0454, 0x12C
int	32 bit	-2.147.483.648 ÷ 2.147.483.647	es. 50000, 041520, 0xC350
long	64 bit	-9.223.372.036.854.775.808 ÷ +9.223.372.036.854.775.807	es. 45L
float	32 bit	± 1,4E-45 ÷ 3,45E+38 7 decimali significativi	es. 3.0F
double	64 bit	± 4,9E-324 ÷ 1,7E+308 15 decimali significativi	es. 3.0
char	16 bit	\u0000 ÷ \uFFFF (set unicode)	es. ' a' , ' \n'
boolean	1 bit		true o false

N.B.:

i letterali ottali ed esadecimali iniziano con 0 e 0x rispettivamente
per i numeri in virgola mobile, un letterale è double per default
per i letterali char esistono *sequenze di escape*

7. Operatori

Operatore	Note
+, -	somma e sottrazione
*, /	moltiplicazione e divisione
%	modulo
++	incremento: post e pre
--	decremento: post e pre
+=, -=, /=, *=, %=	con assegnazione, es. $x+=y$ equivale a $x=x+y$
<, >, <=, >=, ==, !=	confronti
&&, , !	operatori logici (and, or, diverso)
&, , ^	^ è lo XOR
instanceof	
?:	

N.B.:

Alcuni operatori possono essere unari, binari o entrambi
Si possono applicare a tipi elementari o alcuni agli oggetti
Sono stati omessi gli shift

8. Dichiarazioni

8.1 Dichiarazione di classe

```
[public][final|abstract] class <nome della classe>  
[extends <nome classe>]  
[implements <interfaccia[, interfaccia]>]  
{  
[variabili, metodi, inner classes]  
}
```

L' ordine dei primi due modificatori è facoltativo.
Esempio:

```
class Minimal {}  
abstract public Tester{}
```

8.2 Dichiarazione di interfaccia

```
[public] interface <nome interfaccia>[extends <interfaccia>[,interfaccia]]  
{  
[variabili dichiarate implicitamente o esplicitamente public static final, metodi public senza altri  
modifier]  
}
```

N.B.:

una interfaccia può estendere (*extends*) altre interfacce.

8.3 Dichiarazione di metodi

```
public|private|protected|<none> [abstract|final] [native][static][synchronized]  
<return type>/void <nomeMetodo>([<arguments list>]) [throws <lista eccezioni>]
```

L' ordine dei modifier prima del tipo di ritorno e del nome è ininfluente.

8.4 Dichiarazione del main

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

args ovviamente può essere un identifier valido qualunque.

8.5 Dichiarazione del costruttore

Il costruttore è un metodo che:

- x Ha il nome della classe
- x Non dichiara tipo di ritorno
- x Non può essere final, abstract, synchronized, native o static.

- x Se nessun costruttore è dichiarato, il metodo privo di parametri è disponibile implicitamente.
- x La prima istruzione è, implicitamente o esplicitamente con il richiamo del costruttore della super classe con *super()* a meno che non sia ridefinita con *super(parametri)*.
- x All'interno può fare riferimento ad altri costruttori con l'istruzione *this(parametri eventuali)*.

8.6 Dichiarazione ed inizializzazione di variabili

Variabili membro

```
public|private|protected|<none> [final][static][transient][volatile]
<tipo> <identificatore>
```

Vengono inizializzate implicitamente con i valori:

- x zero per i numerici
- x false per i boolean
- x \u0000 per i char
- x null per altri oggetti o array (con conseguenti eventuali runtime error)

Variabili locali o automatiche

```
<tipo> <identificatore>
```

Devono essere esplicitamente inizializzate o non compila.

9. Convenzioni nomenclatura

In genere non si utilizza il simbolo `_` per separare parole, tranne che per le costanti.

- x **Classi** – Generalmente sono sostantivi con la lettera iniziale maiuscola. Se il nome della classe è composto da più parole unite, allora ogni parola aggiunta ha l' iniziale maiuscola. Esempi: Controllore, ListaDiValidazione.
- x **Interfacce** – Come per le classi.
- x **Metodi** – Generalmente sono verbi con la lettera iniziale minuscola. L' unione di più parole viene realizzata unendo la parola con l' iniziale maiuscola. Inoltre, per la convenzione Java Bean, esistono 3 prefissi: get, set, is. Esempi: disegnaBordo(), aggiungiOspite(), getNome(), setIndirizzo(), isDisponibile().
- x **Costanti** – Tutto in maiuscolo, con `_` come separatore. Esempi: PI_GRECA, VALIDA, RISPOSTA_ESATTA.
- x **Variabili** – Lettera iniziale minuscola, parole giunte con iniziale maiuscola. Esempi: contatore, rispostaUtente, i,j.

10. Visibilità di classi e metodi

Vengono di seguito descritte come il modificatore della visibilità opera a livello di classe e di membri (attributi o metodi).

10.1 Visibilità di classe

Pubblica (public) o default (nessun modifier): visibile fuori dal package o solo nel package.

10.2 Visibilità di membri (attributi o metodi)

Modifier	Istanza nel package	In package differenti	Sottoclasse nel package	Sottoclasse fuori dal package
public	sì	sì	sì	sì
private	no	no	no	no
protected	sì	no	sì	sì
default	sì	no	sì	no

11. Controllo di flusso

11.1 statement di scelta: if-then-else

```

If(<condizione>)
  <statement1>
[else
  <statement2>]

```

dove :

- x <condizione> è una condizione booleana semplice - esempio `importo>100` - o un' espressione - esempio `importo>100 && eta>20`
- x <statement1> e <statement2> sono istruzioni semplici - esempio `System.out.println("Hello");` - o composte `{reddito=calcolaReddito(this, eta); reddito.stampaRisultati()}`
- x else è opzionale, come indicato dalle parentesi quadre.

Esempio:

```

if(persona!=null && persona.eta>18)
  System.out.println("Maggiorenne");
else{
  System.out.println("Minorenne");
}

```

```

    forward(homePage);
}

```

11.2 Statement di scelta: switch-case-default

```

switch( <variabile di selezione>){
    case <valore 1>:
        <statement 1>
        [break;] // statement java per uscire dal case
    case <valore 2>:
        <statement 2>
        [break;] // statement java per uscire dal case
    .....
    case <valore n>:
        <statement n>
        [break;] // statement java per uscire dal case
    [
    default:
        <statement default>
    ]
}

```

dove:

- x <variabile di selezione> è la variabile che determina l' esecuzione del corrispondente ' caso' . Essa deve essere intera o carattere (non String o altri tipi).
- x <valore 1> .. <valore n> sono i valori assegnati a ciascun caso: quando la variabile di selezione possiede uno di questi valori, allora il corrispondente statement ed i successivi sono eseguiti. Gli statement successivi al case corrispondente non vengono eseguiti se presente l' istruzione `javabreak`.
- x Se nessun valore è elencato nei ' casi allora viene eseguito lo stement default se presente un caso con tale dichiarazione.

Esempio:

```

char tipoPatente;
tipoPatente=leggiTipoPatente();
switch(tipoPatente) {
    case 'A':
        System.out.println("Puoi guidare la moto");
        break;
    case 'B':
        System.out.println("Puoi guidare l'auto");
        break;
    // mancano gli altri tipi ...
    default:
        System.out.println("Non puoi guidare niente");
}

```

11.3 Statement di iterazione: for

```
for(<inizializzazione>; <condizione>; <incrementi>)  
  <statement>
```

dove:

- x <inizializzazione> è il blocco di statement eseguiti prima di iniziare il ciclo, una sola volta- Esempio: $i=0$
- x <condizione> è una condizione o un' espressionebooleana. Viene valuta ogni volta prima di iniziare una iterazione e se la condizione è vera allora inizia il ciclo - Esempio: $i<10$.
- x <incrementi> è il blocco di statement che viene utilizzato al termine di ogni iterazione.
- x <statement> l' istruzione o il blocco di istruzioni iterati.

Esempio:

```
for(int i=0, j=10; i<10;i++,j--)  
  rovesciati[j]=elementi[i]; // rovescia l' array
```

11.4 statement di iterazione: while

Versione top tested

```
while(<condizione>)  
  <statement>
```

versione bottom tested

```
do  
  <statement>  
while(<condizione>)
```

dove:

- x <condizione> è una condizione o un' espressionebooleana che viene valuta prima di iniziare le iterazioni
- x <statement> l' istruzione o il blocco di istruzioni iterati.
- x La differenza tra la versione top e bottom tested è che nel secondo caso lo statement viene eseguito almeno una volta

Esempio:

```
while(iteratore.hasNext())  
  ((Risultati)iteratore.next()).stampa(); // prima next, poi cast, poi stampa  
  
do{  
  fineFile=leggiRecord();  
  stampaRisultato()  
} while(!fineFile)
```

12. Gestione delle Exception

Java consente la gestione delle eccezioni nel flusso principale con il costrutto

```
try{
// Codice del flusso principale
}
catch(Exception ex){
// Codice per il rilascio delle risorse, ecc.
}
finally{
// Codice che viene sicuramente eseguito
}
```

- x Le clausole try-catch-finally non possono ammettere istruzioni intermedie
- x Possono essere presenti più catch.
- x Le exception nella catch possono appartenere a tipi e sottotipi. Java applica l' exception matching: se voglio trattare un sottotipo in un modo ed il tipo in un altro bastano due catch
- x La clausola catch può essere assente se presente finally
- x L' exception deve essere gestita se di tipo checked, non necessariamente deve essere gestita se di tipo RuntimeException
- x L' overriding di metodi che generano eccezioni non necessariamente devono generare eccezioni dello stesso tipo; sono consentite eccezioni di sottotipi
- x E' possibile rigenerare una eccezione modificando lo stackTrace. Vedi esempio.

Esempio: rigenerazione di una exception modificando lo stack trace.

metodo1 genera una exception
metodo2 richiama metodo1

Primo caso:

```
public void metodo2() throws java.lang.Exception {
    metodo1();
}
```

Lo stack trace si presenta così:

```
java.lang.Exception: Eccomi qua
java.lang.Throwable(java.lang.String)
java.lang.Exception(java.lang.String)
void appunti.Exception1.metodo1()
void appunti.Exception1.metodo2()
void appunti.Exception1.main(java.lang.String [])
```

Secondo caso:

```
public void metodo2() throws java.lang.Exception {
    try {
```

```
        metodo1();
    } catch (Exception ex) {
        Throwable ce = ex.fillInStackTrace();
        throw (Exception) ce;
    }
}
```

Lo stack trace si presenta così:

```
java.lang.Exception: Eccomi qua
void appunti.Exception1.metodo2()
void appunti.Exception1.main(java.lang.String [])
```

13. La Garbage Collection

La garbage collection è l'operazione automaticamente gestita da Java per la rimozione degli oggetti inutilizzati (non più referenziati) ed ottimizzare la memoria disponibile.

La garbage collection non deve essere necessariamente gestita dal programmatore. Java consente comunque di invocare il garbage collector con il metodo `System.gc()` o l'analogo nella classe `Runtime`; la JVM non garantisce che ciò avvenga.

13.1 Algoritmi di garbage collection

L'attività di garbage collection è una attività onerosa e deve essere ottimizzata. Un algoritmo di implementazione è denominato `mark - sweep`. Funziona in due passaggi per elaborazione:

- x `MARK`. Identificazione degli oggetti non referenziati.
- x `SWEEP`. Eliminazione degli oggetti identificati dalla fase di `SWEEP` della elaborazione precedente.

Inoltre, alcune implementazioni di garbage collector, definite *generazionali*, ad ogni passaggio determinano se l'oggetto è stato promosso (operazione di *promoting*) ovvero è già sopravvissuto ad altri passaggi di individuazione degli oggetti non referenziati; gli oggetti promossi non saranno in seguito più oggetto di controllo nelle successive 10 elaborazioni.

L'algoritmo generazionale avrà quindi un numero ridotto di oggetti da controllare, migliorando quindi le proprie performance.

13.2 Finalize

Non è possibile determinare a priori quando un oggetto viene distrutto se non nel momento in cui ciò avviene mediante l'utilizzo del metodo `finalize()`.

Il metodo `finalize()` - signature `public void finalize()` - viene richiamato dal garbage collector quando l'oggetto sta per essere distrutto.

13.3 Referenze

Normalmente un oggetto possiede un handle che mantiene la referenza ad un oggetto. L' handle viene generato con l' istruzione `new` seguita dal costruttore della classe. Questo tipo di referenza viene mantenuta finchè l' oggetto viene utilizzato (più precisamente fino al termine del blocco di codice che ha la visibilità sull' oggetto).

Quando l' oggetto non ha più referenze è pronto per la garbage collection.

Con Java 2 sono state rese disponibili delle classi per mantenere diversi tipi di referenze più deboli. Una referenza debole non viene garantita ma talvolta ciò può essere utile per rendere l' applicazione più snella o garantirsi che gli oggetti referenziati con questi meccanismi vengano distrutti.:

SoftReference	Se il GC ha a disposizione abbastanza memoria libera, allora mantiene la soft reference altrimenti distrugge l' oggetto (ovviamente se non ha referenze forti). Utile per meccanismi di caching.
WeakReference	Il GC distrugge sicuramente l' oggetto (sempre se non referenze forti). Utile per controllare se un oggetto è ancora in uso, ad esempio la lista di finestre aperte in una applicazione.
PhantomReference	Il GC richiama il finalize dell' oggetto, lo accoda (vedi in seguito) ma non libera la memoria. Utilizzato per oggetti gestiti anche da altri ambienti attraverso Java Native Interface (un programma C++, ad es.). La memoria deve essere rilasciata richiamando il clear dell' oggetto in coda.

Una *Reference* mette a disposizione i metodi:

get()	Ritorna una referenza all' oggetto, se ancora esistente altrimenti <i>null</i> . Per <i>PhantomReference</i> torna sempre <i>null</i> .
clear()	Elimina la referenza. Dopo la clear, <i>get</i> ritorna sempre <i>null</i> .
enqueue()	Accoda l' oggetto
isEnqueued()	Ritorna true se l' oggetto è stato accodato.

Esempio:

```
SoftReference previousImage = new SoftReference(myObject, null); // null è la coda
```

Una referenza debole può essere associata ad una *ReferenceQueue* - coda di referenza. Le *Phantom* sono sempre associate ad essa. Il meccanismo di GC prima accoda una referenza e poi distrugge l' oggetto (tranne le *Phantom*).

La coda può essere un meccanismo per conoscere o intercettare l' evento di distruzione dell' oggetto.

Metodi definiti da *ReferenceQueue*:

poll()	Ritorna una referenza se l' oggetto è in coda. <i>null</i> se non è presente in coda
remove()	Ritorna una referenza se l' oggetto è in coda e lo rimuove dalla coda. <i>null</i> se non è presente in coda.
remove(long delay)	Come il precedente ma con un delay sulla rimozione.

14. Encapsulation in OOD

Encapsulation e information hiding sono due aspetti molto importanti nell' ObjectOriented Design. In questo contesto vengono affrontati alcuni aspetti correlati ad OOD finalizzati all' esame di certificazione, meritando l' argomento una ben più ampia trattazione.

14.1 Benefici dell' OOD

- x Manutenibilità. Gli interventi di manutenzione hanno un impatto limitato o nullo al di fuori dell' area di implementazione.
- x Estensibilità. La modularizzazione del software consente di estenderne le funzionalità.
- x Chiarezza. Il codice risulta maggiormente razionalizzato e comprensibile.

14.2 Concetti base sulle relazioni tra oggetti

Esistono due relazioni base:

- x Estensione. Risponde alla domanda *is a* ovvero *è un*.
- x Aggregazione o composizione. Risponde alla domanda *has a* ovvero *ha un*.

15. Ridefinizioni dei metodi

Un metodo può essere ridefinito con due modalità:

- x Un metodo che ha lo stesso nome ma numero e/o tipi di parametri differenti viene detto *overloaded*.
- x Un metodo di una sottoclasse che ha lo stesso nome, lo stesso numero e tipo di parametri, lo stesso return type è detto *overridden*.

15.1 Metodi overridden

Regole per la validità

- x Visibilità: un metodo ridefinito non può che **estendere** la visibilità del metodo dell' avo. Quindi *urprivate* può essere ridefinito *public* ma non viceversa.
- x Exception: un metodo ridefinito può ridurre e non aumentare il set di exception

- generate, nè le eredita. Esempio, il genitore genera Exception il figlio genera URLMalformedURLException che è un subset di Exception.
- x Synchronized. E' indipendente e non influente.

Tipo sottostante

Nel caso di dichiarazione di un tipo e costituzione con un altro, in fase di esecuzione vale l' implementazione del tipo di costituzione.

Esempio:

la classe ClassB estende ClassA. Il metodo *stampa* stampa ' A' per la classe ClassA e ' B' per la classe ClassB.

Cosa succede nel caso di:

```
ClassA obj=new ClassB();  
obj.stampa();
```

Il risultato è la stampa di ' B' .

15.2 Metodi Overloaded

Regole per la validità:

Dichiarazione. Il metodo dichiarato deve avere parametri differenti in numero e/o tipo.

Utilizzo. Il richiamo di un metodo deve essere non ambiguo.

Esempio:

```
void stampa(int i, long l)  
void stampa(long i, int l)
```

Il codice, salvo errori, è compilabile.

Se aggiungo un richiamo la seguente riga di codice

```
stampa(1,2);
```

NON compila perchè ambiguo.

15.3 Costruttori overloaded e overridden

Valgono le stesse regole per i metodi; in aggiunta ne esistono anche delle altre:

- x `this(parametri)`. Per richiamare un costruttore nell' ambito della stessa classe.
- x `super(parametri)`. Per richiamare un costruttore della superclasse
- x `super()` senza parametri viene richiamato implicitamente come prima istruzione di ogni costruttore a meno che non venga richiamato esplicitamente un costruttore con parametri (NB: una superclasse senza costruttore privo di parametri, sottoclasse con costruttori che non iniziano con `super` con parametri -> non compila).

16. Inner classes

Le inner classes sono classi definite all' interno di altre classi, di loro metodi o di blocchi di codici. L' obiettivo è ridurre la visibilità o di ridurre la visibilità alla classe, Ne esistono di diverse tipologie di seguito descritte.

16.1 Static inner classes

- x Dichiarazione: *static class* con eventuale modificatore *public*
- x Eventuale *extends* o *implements*
- x Visibilità: solo a variabili di classe pubbliche e private.

16.2 Member inner classe

- x Dichiarazione: *class* con eventuale modificatore *public*
- x Eventuale *extends* o *implements*
- x Visibilità: variabili membro pubbliche e private
- x Non possono avere membri statici
- x Nota: sintassi particolare per la costruzione. Es: `this.new ClasseInterna();`

16.3 Local inner classes

- x Dichiarazione: all' interno di un metodo della classe esterna, con clausola *class* SENZA modificatore *public*
- x Eventuale *extends* o *implements*
- x Visibilità: variabili locali o parametri *final*
- x Non possono avere membri statici.

Trucco : per accedere all' oggetto contenente si usa la sintassi `ClasseEsterna.this`.

16.4 Anonymous inner classes

- x Sono local inner classes senza nome

- x Dichiarazione: `new ClassePadre(){}`. ClassePadre potrebbe anche essere una interfaccia.
- x Non è possibile usare `extends` o `implements`
- x Visibilità: variabili locali o parametri `final`
- x Non possono avere membri statici
- x Non possono avere costruttori
- x I file generati sono chiamati `ClasseEsterna$1.class, ClasseEsterna$2.class`, ecc anche se la classe anonima fosse definita dentro un' altra inner class.

Trucco 1: per evitare lo static ovvero accedere ad oggetti locali o parametri dichiarati `final` è possibile usare oggetti contenitori o array. Essi non vengono modificati ma il loro contenuto sì.

Trucco 2: non hanno costruttori ma si possono iniziare variabili con il metodo della lazy initialization (`Object getProperty{ if(prop==null) { ... inializza prop}; return prop}`) o con l' iniziatore anonimo (un blocco di codice tra parentesi graffe)

17. Collections

Le collections o containers sono le raccolte di oggetti.

In Java 1 erano disponibili Vector ed Hashtable; in Java 2 sono disponibili interfacce ed implementazioni maggiormente sofisticate.

Esistono due gerarchie, Collection e Map:

Collection (interface)

^

|_____ Set (interface)

^

|_____ SortedSet (interface)

^

|_____ List (interface)

Map (interface)

^

|_____ SortedMap (interface)

I Vector sono assimilabili a List, Hashtable sono assimilabili a Map

17.1 Definizione di Collection

```
public interface Collection {
    // Operazioni di base
    int size();
}
```

```

boolean isEmpty();
boolean contains(Object element);
boolean add(Object element); // Optional
boolean remove(Object element); // Optional
Iterator iterator();

// Operazioni di gruppo
boolean containsAll(Collection c);
boolean addAll(Collection c); // Optional
boolean removeAll(Collection c); // Optional
boolean retainAll(Collection c); // Optional
void clear(); // Optional

// Conversioni in array
Object[] toArray();
Object[] toArray(Object a[]);
}

```

L' iteratore di una collection (analogo ad Enumeration dei Vector):

```

public interface Iterator {
    boolean hasNext();
    Object next();
    void remove(); // Optional
}

```

17.2 Definizione di Set

```

public interface Set {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element); // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c); // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear(); // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}

```

17.3 Definizione di List

```

public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element); // Optional
    void add(int index, Object element); // Optional
    Object remove(int index); // Optional
    abstract boolean addAll(int index, Collection c); // Optional
}

```

```
// Search
int indexOf(Object o);
int lastIndexOf(Object o);

// Iteration
ListIterator listIterator();
ListIterator listIterator(int index);

// Range-view
List subList(int from, int to);
}
```

Iteratore di lista:

```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

    void remove(); // Optional
    void set(Object o); // Optional
    void add(Object o); // Optional
}
```

17.4 Definizione di Map

```
public interface Map {
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

17.5 Sintesi su Collection

Interfaccia	Implementazione	Descrizione
Set	HashSet	Non ammette duplicati, basata su Hashtable.
	TreeSet	Memorizza in un albero, mantiene l' ordine.
List	ArrayList	Analogo a Vector
	LinkedList	Analogo a Vector
SortedSet		
Map	HashMap	Analogo a Hashtable, basato su Hashtable
	TreeMap	Analogo a Hashtable, basato su albero.
SortedMap		

17.6 Wrapper Implementation

Delegano alle specifiche collezioni, ma aggiungono servizi. Utilizzano il pattern delle factory utilizzando metodi statici, tutti raccolti in Collections. Tra i metodi offerti c' è sort. Ad esempio, per ordinare una lista l: Collections.sort(l).

Convenience Implementation

Raccolta di metodi di convenienza:

- x Arrays.asList. Esempio: List l = Arrays.asList(new Object[size]);
- x Collections.nCopies. N copie dello stesso oggetto, come lista. Esempio: List l = new ArrayList(Collections.nCopies(1000, null)); // 1000 oggetti null .
- x Collections.singleton. Una lista di un solo specifico oggetto. Esempio: profession.values().removeAll(Collections.singleton(LAWYER));
- x Collections.EMPTY_SET, Collections.EMPTY_LIST. Costanti.

18. Threads

18.1 Creazione

- x Classe che estende Thread. Ridefinizione del metodo public void run().
- x Classe che implementa interfaccia Runnable

18.2 Avvio

- x istanzaThread.start() nel primo caso
- x new Thread(istanzaRunnable).start() nel secondo caso

18.3 Sleep e interrupt

- x Thread.sleep(long millisec) throws InterruptedException. Porta il thread in uno stato di inattività
- x interrupt() setta il flag omologo e se l' oggetto è in sleep lo interrompe e genera una eccezione InterruptedException, altrimenti previene che ciò avvenga, sempre per exception.
- x Thread.interrupted() legge e pulisce il flag
- x isInterrupted() legge il flag

18.4 Concorrenza

Il modifier *synchronized* blocca l' oggetto al quale viene richiesta l' esecuzione o blocca il richiamo se tale oggetto è già bloccato da altri thread.

Si può bloccare una parte di codice o un altro oggetto usando la sintassi

```
synchronized(this o altri oggetti) { .. codice thread safe .. }
```

La gestione di processi di tipo produttore-consumatore può essere effettuata con wait(), notify() o notifyAll() della classe Object. Tali metodi, se non eseguiti dentro blocchi thread safe, generano un unchecked exception IllegalMonitorStateException.

18.5 Scheduling

- x Thread.yields() consente di far perdere il proprio turno di CPU time. Produce comunque overhead e quindi un uso eccessivo può peggiorare le performance.
- x Per un thread è possibile definire la priorità, da 1 (MIN_PRIORITY) a 10 (MAX_PRIORITY). La priorità normale è 5 (NORM_PRIORITY).

18.6 ThreadGroups, security e setDaemon()

- x Un Thread è sempre associato ad un ThreadGroup per operazioni di gruppo. La priorità del gruppo non può essere superata.
- x Il SecurityManager può gestire il setPriority
SetDaemon(true) setta il daemon flag. Il thread viene fermato quando la JVM rileva solo daemon thread.

19. Bibliografia e riferimenti

[1] Sun Certified Programmer for Java 2 - Syngress / OSborne

[2] <http://java.sun.com/docs/books/tutorial/index.html> - Il tutorial Java di SUN

[3] <http://java.sun.com/docs/index.htm> - Documentazione JAVA SUNI

20. Marchi

JSP™ e JAVA™ sono marchi registrati di SUN MICROSYSTEMS.

Apache, Tomcat e Jakarta sono marchi di The Apache Software Foundation

IBM™, Microsoft™ e Oracle™ sono marchi registrati dai legittimi proprietari.