



jAPS 2.0 – Linee guida e standard di codifica

jAPS 2.0 a partire dalla Versione 2.0



Autori

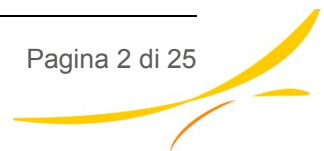
Eugenio Santoboni

Copyright © 2009 – Agiletec s.r.l.

Il presente documento è pubblicato sotto licenza GNU Free Documentation License.

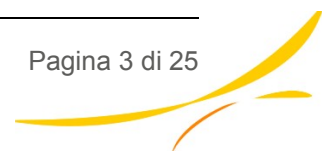
L'uso e la distribuzione del documento, o delle informazioni in esso contenute, è consentito secondo i termini della licenza [FDL GNU Free Documentation License](#).

Versione documento 1.2 del 9-Febbraio-2009.

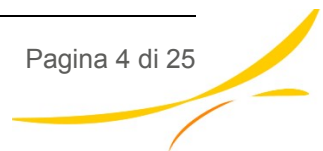


Indice

1	Scopo del documento.....	5
1.1	Introduzione.....	5
1.2	A chi è destinato.....	5
1.3	Prerequisiti.....	5
1.4	Riferimenti.....	5
2	Cosa è jAPS.....	6
3	Introduzione.....	7
3.1	Convenzioni del documento.....	7
3.2	Contenuti del documento.....	7
3.3	Perché avere uno standard di codifica.....	7
4	Documentazione del sistema.....	8
4.1	Scelta dei nomi.....	8
4.1.1	Nomi delle classi.....	9
4.1.2	Nomi delle variabili d'istanza, di classe e locali ai metodi.....	9
4.1.3	Nomi dei metodi.....	10
4.1.4	Nomi degli argomenti dei metodi.....	11
4.2	Documentazione.....	11
4.2.1	Documentazione delle classi.....	11
4.2.2	Documentazione dei metodi.....	12
4.2.3	Documentazione aggiuntiva.....	12
4.2.4	Convenzioni di formattazione.....	12
4.2.4.1	Uso degli spazi.....	12
4.2.4.2	Metodi, indentazioni e righe vuote.....	13
4.2.4.3	Lunghezza espressioni.....	13
5	Convenzioni di codifica.....	17
5.1	Inizializzazione variabili di istanza.....	17
5.2	Inizializzazione variabili locali.....	17
5.3	Accesso a variabili e metodi static.....	18
5.4	Accesso alle variabili d'istanza.....	18
5.5	Getters per le costanti.....	19
5.6	Metodi accessori per le collezioni.....	19
5.7	Utilizzo dei parametri dei metodi.....	19
6	Linee Guida Generali Specifiche progetto jAPS2.....	21



7 Errori comuni e suggerimenti.....24



1 Scopo del documento

1.1 Introduzione

Lo scopo di questo documento è fornire le linee guida e gli standard di codifica utilizzati nel progetto jAPS 2.0.

L'obiettivo è quello di migliorare la leggibilità del codice sorgente di jAPS avvalendosi di una serie di best practices di scrittura riconosciute come standard nel mondo object oriented.

1.2 A chi è destinato

Questo documento è destinato agli sviluppatori che intendono sviluppare nuove funzionalità o servizi applicativi con jAPS.

1.3 Prerequisiti

Per poter utilizzare efficacemente le informazioni contenute in questo documento, occorre avere le basi di programmazione object oriented ed esperienza sulla piattaforma Java.

1.4 Riferimenti

Ulteriori informazioni possono essere richieste scrivendo alle mailing-list:

- ✓ japs-devs@lists.sourceforge.net per la mailing-list dedicata agli sviluppatori;
- ✓ japs-users@lists.sourceforge.net per la mailing-list dedicata ai semplici utilizzatori.

2 Cosa è jAPS

jAPS 2.0 è una piattaforma Open Source, professionale, per la realizzazione di Portali Internet e Intranet di tipo informativo, collaborativo e di servizi. La piattaforma nasce in Italia in linea con le normative vigenti ed è stata progettata per soddisfare le esigenze di aziende e Pubbliche Amministrazioni.

jAPS 2.0 consente di creare portali “su misura”, accessibili secondo i requisiti di accessibilità definiti nell'Allegato A – D.M. 8 Luglio 2005 (relativo alla Legge Stanca) e semplici da usare anche per utenti non tecnici.

La piattaforma jAPS 2.0 è costituito da due componenti principali:

- **jAIF (Java Agile Integration Framework)**

- un framework progettato per fornire un insieme di componenti di base per sviluppare Portali “su misura”, flessibili e accessibili e per poter integrare servizi applicativi

- **jACMS**

un Web Content Management System, integrato in jAIF, che rispetta i requisiti di accessibilità definiti nell'Allegato A – D.M. 8 Luglio 2005 (relativo alla Legge Stanca). jACMS consente di gestire contenuti in maniera intuitiva, flessibile e semplice.

1 Introduzione

1.1 Convenzioni del documento

Gli statement sono indicati in grassetto, e preceduti da un quadratino:

- ✓ **Questo è uno statement.**

Gli esempi di codice sono indicati in carattere corsivo:

- ✓ *esempio di codice Java corretto*

Gli esempi di codice non corretto sono indicati come gli esempi di codice, ma barrati:

- ✓ ~~esempio di codice Java non corretto~~

1.1 Contenuti del documento

Il presente documento riporta le linee guida da seguire per la documentazione del software prodotto.

L'obiettivo è quello di migliorare la leggibilità del codice del progetto jAPS avvalendosi di una serie di best practices di scrittura riconosciute come standard nel mondo object oriented.

Il codice dovrà essere il più possibile “autoesplicativo”.

1.2 Perché avere uno standard di codifica

Le convenzioni sul codice sono utili ai programmatori per i seguenti motivi:

- ✓ 80% del costo sul ciclo di vita del software viene speso in manutenzione;
- ✓ difficilmente il codice viene gestito sempre dal suo autore;
- ✓ convenzioni sul codice aumentano la leggibilità del software permettendo una comprensione di codice nuovo molto più veloce.

1 Documentazione del sistema

1.1 Scelta dei nomi

- ✓ I nomi di tutti gli elementi del sistema devono essere scritti in Inglese.
- ✓ Utilizzare una terminologia applicabile e inerente al dominio.

Se per esempio i vostri utenti si riferiscono alla pagina web su cui è possibile visualizzare i contenuti come “pagina” allora utilizzare il nome di classe “Page”

- ✓ Scegliere nomi descrittivi, anche se lunghi. Evitare comunque nomi troppo lunghi. Sebbene nomi come `nPag`, `contPag` e `titPag` siano facili da scrivere non forniscono indicazioni su ciò che rappresentano e, inoltre, risultano essere non immediatamente leggibili:

pageNumber, pageContent, pageTitle

~~*nPag, contPag, titPag*~~

- ✓ Evitare di utilizzare il più possibile abbreviazioni e se vi è la necessità usarle in maniera opportuna e comunque sempre la stessa:

content → *cont*

- ✓ Mettere in maiuscolo la prima lettera di acronimi standard. Per esempio se si utilizza la parola SQL, nomi come `sqlDatabase` per un attributo o `SqlDatabase` per una classe sono più facili da leggere di `sQLDatabase` o `SQLDatabase`:

sqlDatabase

SqlDatabase

sQLDatabase

SQLDatabase

- ✓ Usare nomi con lettere minuscole per le variabili locali, di istanza e per i metodi mettendo in maiuscolo la prima lettera delle parole di un nome composto eccetto la prima:

pageTitle, sendMessage, _instanceVariableName

- ✓ Usare nomi inglesi, Utilizzare i seguenti termini inglesi che fanno parte di un vocabolario di termini di uso comune:

create, destroy, load, save, get, set, put, do, perform, find, insert, select, update, delete, query, print, store, is, has

1.1.1 Nomi delle classi

- ✓ Scegliere nomi che descrivono un oggetto della classe, al singolare:

Page, Content

1.1.1 Nomi delle variabili d'istanza, di classe e locali ai metodi

- ✓ Le variabili di istanza devono essere precedute da un underscore “_” per poter essere meglio localizzate all’ interno del codice:

_instanceVariableName, _page

- ✓ Scegliere nomi che descrivono il ruolo della variabile, non il tipo:

fileName, selectedValues, pageReference,

dataCollection, ~~pageArray~~

- ✓ Nel caso di variabili Booleane, scegliere predicati o aggettivi:

contentsReserved, _active, onlySelectedPages

- ✓ Le costanti in java sono implementate come “static final” devono essere scritte in maiuscolo e se rappresentano un nome composto devono utilizzare il simbolo “_” per separare le diverse unità che compongono il nome:

ACTIVE, NAME_ELEMENT

- ✓ Le collezioni, come array o vettori etc., dovrebbero avere un nome al plurale che indichi il tipo di oggetti registrati:

pages

_orders

- ✓ Non nascondere i nomi. Evitare di dare un nome ad una variabile con scope inferiore uguale o simile a quello di scope maggiore:

Ad esempio se si è chiamato “page” un parametro è meglio non creare una variabile locale che si chiami “page”

- ✓ VARIABILI LOCALI

E' importante dare nomi opportuni a :

- Streams: *in, out, inOut*
- Contatori: *i,j,k*
- Exceptions: *e,ex*

Dichiarare sempre una sola variabile per riga

- ✓ Usare una variabile locale solo per un unico scopo

1.1.1 Nomi dei metodi

- ✓ I nomi dei metodi iniziano sempre con lettera minuscola.
- ✓ Costruttori: i costruttori si occupano di effettuare le necessarie inizializzazioni quando un oggetto viene creato. Il costruttore ha sempre il nome della classe.
- ✓ Scegliere verbi e frasi al modo imperativo per i metodi che effettuano azioni

verify, safeOnFile, readFromFile, execute

- ✓ Scegliere nomi che permettano di leggere l'invocazione del metodo come una frase italiana

readFromFile(fileName)

insertFileName(fileName)

- ✓ I nomi composti presentano la prima lettera della nuova parola in maiuscolo

modifyFileName()

- ✓ Nel caso di metodi di interrogazione di stato che rendono un Boolean, usare una frase che inizia con *is* oppure *has*

isOpen, hasMoreThanOneValue

- ✓ I metodi che leggono o impostano una variabile d'istanza non Boolean hanno il nome della variabile stessa, con il prefisso *get* per quelli che recuperano un valore e *set* per quelli che settano un valore.

Se “name” e “file” sono due variabili di istanza i relativi metodi *setters* e *getters* saranno:

getName, setName, getFile, setFile

readName

1.1.1 Nomi degli argomenti dei metodi

- ✓ E' importante dare informazione sul tipo dell'argomento:

insert(String prefix), add(Point point)

- ✓ Nel caso di argomenti multipli dello stesso tipo, usare il nome del tipo con una qualificazione di ruolo:

joinPoints(Point upper, Point left, Point right)

- ✓ Non inserire nel nome l'indicativo del tipo:

insertUsers(List<User> userList)

insertUsers(List<User> users)

1.1 Documentazione

1.1.1 Documentazione delle classi

- ✓ Per ogni classe, scrivere una descrizione della medesima secondo lo standard javadoc

Il javadoc principale deve essere redatto in inglese e deve fornire indicazione

quali versione (@version) , autore (@author) ed eventuali altre informazioni (su tag consentiti).

L'informazione del Copyright deve essere inserita all'inizio della classe prima del riferimento del package.

1.1.1 Documentazione dei metodi

- ✓ In linea di principio, i metodi devono essere brevi (max. 10-15 righe) e senza commenti interni al codice.
- ✓ E' necessario scrivere una descrizione del metodo secondo lo standard javadoc.

1.1.1 Documentazione aggiuntiva

- ✓ Possono essere creati diagrammi e documentazione UML che descrivono graficamente le classi e le loro relazioni avvalendosi di strumenti automatici.

1.1.1 Convenzioni di formattazione

Si tratta di convenzioni per rendere uniforme e quindi più leggibile il codice Java. Esse comprendono linee guida su come scrivere singole espressioni e blocchi, su come scrivere istruzioni condizionali e di iterazione.

1.1.1.1 Uso degli spazi

- ✓ Mettere uno spazio dopo la virgola (“,”) nelle definizioni dei parametri dei metodi.

openFile(String fileName, String path).

openFile(String fileName,String path).

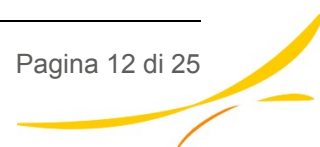
- ✓ Mettere uno spazio prima e dopo gli operatori binari: + - * / < = > | == <= >=, e l'assegnazione =.

*result = (2 + 3) / (7 * 7).*

*result=(3+4)/(5*7).*

- ✓ Usare gli spazi nel codice

result = customerCash + supplierCash



result=customerCash+supplierCash

1.1.1.1 Metodi, indentazioni e righe vuote

- ✓ Lasciare almeno una riga vuota tra i metodi (e costruttori) inseriti.
- ✓ Il nome del metodo va scritto con indentazione di un tab (4 spazi) rispetto al bordo della pagina. Tutte le altre righe devono essere indentate di un “Tab” (4 spazi) a partire dal tab precedente. Allo stesso modo, le indentazioni dei blocchi interni del corpo del metodo, devono presentare la indentatura di un “Tab” (4 spazi).
- ✓ Non lasciare righe vuote all'interno del corpo metodo.

1.1.1.1 Lunghezza espressioni

Quando una espressione è troppo lunga per poter stare nel campo di visibilità di una linea occorre suddividerla seguendo i seguenti principi:

- ✓ Interrompere dopo la virgola.
- ✓ Interrompere prima di un operatore:

Esempi:

```
int var1 = this.getValue(longVarName1, longVarName2,  
longVarName3,
```

```
longVarName4, longVarName5);
```

```
int var2 = this.getValue(longVarName1,
```

```
this.getValue2(longExpression1,
```

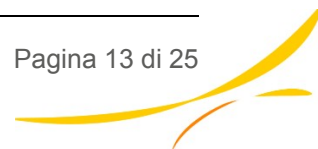
```
longVarName3));
```

- ✓ Di seguito ci sono due esempi su come suddividere un'espressione aritmetica:

```
longVarName = longVarName2 * (longVarName3 + longVarName4)
```

```
+ 4 * longVarName5;
```

```
longVarName1 = longVarName2 * (longVarName3 + longVarName4
```



```
longVarName5) + 4 * longVarName6;
```

- ✓ *Di seguito ci sono due esempi di indentazione di dichiarazione dei metodi:*

//INDENTAZIONE CONVENZIONALE

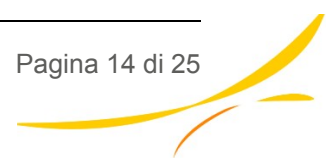
```
unMetodo(int unArg1, Object unArg2, String unArg3,  
Object unArg4) {  
    .....  
}
```

//NON USARE QUESTA INDENTAZIONE con un solo tab a partire
//dall'istruzione if

```
if((condition1 && condition2)  
    || (condition3 && condition4)  
    ||!(condition5 && condition6)) {  
    executeSomethig ();  
}
```

//USARE QUESTA INDENTAZIONE con due tab a partire dall'istruzione
if

```
if ((condition1 && condition2)  
    || (condition3 && condition4)  
    ||!(condition5 && condition6)) {  
    execute();  
    } else {  
    .....  
}
```



- ✓ Qui ci sono tre possibili formati per le espressioni ternarie:

```
alpha = (aLongBooleanExpression) ? beta : gamma;
```

```
alpha = (aLongBooleanExpression) ? Beta
```

```
    : gamma;
```

```
alpha = (aLongBooleanExpression)
```

```
    ? beta
```

```
    : gamma;
```

- ✓ Non mettere mai più di un'istruzione per riga:

```
name = "Mario";
```

```
surname = "Rossi";
```

```
name = "Mario"; surname = "Rossi";
```

- ✓ Usare l'indentazione per evidenziare espressioni condizionali e di ciclo.
- ✓ La parentesi graffa è obbligatoria anche quando all'interno del blocco si ha una sola istruzione.
- ✓ La parentesi graffa di inizio di un blocco deve stare sulla stessa riga del codice che la precede:

```
if (6 == counter) {
```

```
    isOpen = false;
```

```
} else {
```

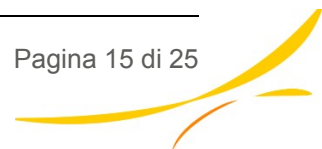
```
    executeSomething();
```

```
}
```

```
if (5 == contatore) isOpen = false;
```

```
while (5 == counter) {
```

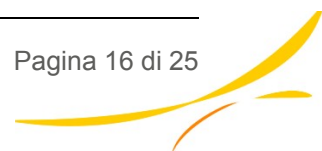
```
    isOpen = false;
```



```
.....  
  
} try {  
    statements;  
} catch (ExceptionClass e) {  
    statements;  
} finally {  
    statements;  
}
```

ESEMPIO

```
public Vector readFromFile(String fileName) {  
    String nome = "";  
    Vector names = new Vector();  
    boolean isNecessary = true;  
    int counter = 0  
    while(isNecessary) {  
        String newName = "Pippo";  
        names.add(newName);  
        if ( counter == 5) {  
            isNecessary = false;  
        }  
        counter += 1;  
    }  
    return names;  
}
```



1 Convenzioni di codifica

1.1 Inizializzazione variabili di istanza

Le variabili di istanza dovrebbero essere inizializzate prima che si acceda ad esse. E' possibile utilizzare i seguenti due approcci per l'inizializzazione:

- ✓ Inizializzare tutte le variabili quando l'oggetto viene creato (approccio tradizionale).

Per questo scopo vengono usati i costruttori.

L'inizializzazione dello stato di un nuovo oggetto a valori "vuoti" ma coerenti è fatto dal metodo d'istanza `inializza`.

Il costruttore chiama il metodo `inializza` (deve essere `protected`) che provvede all'inizializzazione delle variabili di istanza.

- ✓ Inizializzare le variabili la prima volta che vengono utilizzate (lazy initialization).

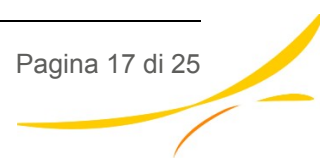
Esempio

```
protected int getMaxValue(){  
  
    if (_maxValue == 0){  
  
        this.setMaxValue(1000);  
  
    }  
  
    return this._maxValue;  
  
}
```

1.1 Inizializzazione variabili locali

- ✓ Cercare di inizializzare le variabili quando vengono dichiarate. L'unica ragione per non inizializzare una variabili è che il suo valore dipenda da qualche computazione che avviene prima.
- ✓ Cercare di inserire le dichiarazioni di variabili all'inizio di un blocco:

```
public void myMethod() {
```



```
int int1 = 0;

.....

if (condition) {

    int counter = 0;

    .....

}

}
```

1.1 Accesso a variabili e metodi static

- ✓ Evitare di accedere a variabili static final o ad un metodo static tramite l'istanza di un oggetto.

1.1 Accesso alle variabili d'istanza

- ✓ Accedere alle variabili d'istanza esclusivamente attraverso i relativi metodi di lettura (getters) e scrittura (setters), mai in modo diretto:

```
String code = this.getCode();

if (code.equals("12")) {

    .....

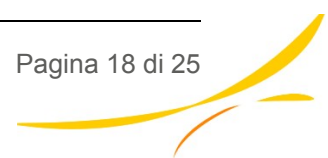
}

if (_code.equals("12")) {.....

_____ _code = "13";

this.setCode("13");
```

- ✓ Le variabili che rappresentano un insieme di oggetti hanno metodi di accesso per:
 - accedere a tutto l'insieme
 - aggiungere un elemento



Possono poi avere metodi per:

- cancellare un elemento
- svuotare l'insieme

1.1 Getters per le costanti

- ✓ L'uso delle variabili statiche è ammissibile solo nel caso in cui il valore della costante sia effettivamente costante nel tempo. Ad esempio la variabile statica `DAYS_IN_A_WEEK` si sa con certezza che non cambierà. In questo caso ha senso lasciare la variabile statica. Tuttavia la maggior parte delle volte il valore della costante può variare per effetto di cambiamento dei requisiti. In tal caso è necessario utilizzare i getters per recuperare il valore della variabile statica.

```
static final int DAYS_IN_A_WEEK = 7
```

```
static int MAX_VALUE = 100
```

1.1 Metodi accessori per le collezioni

- ✓ Le collezioni come array e vettori hanno necessità di avere metodi che facciano le più comuni operazioni di lettura, scrittura, inserimento, cancellazione, creazione:

```
getTickets()
```

```
setTickets()
```

```
addTicket(Ticket ticket)
```

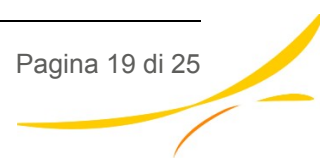
```
removeTicket()
```

```
createTicket()
```

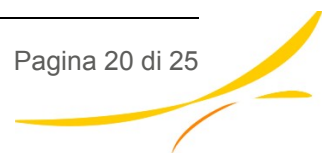
1.1 Utilizzo dei parametri dei metodi

- ✓ I parametri dei metodi non possono essere usati come variabili locali all'interno del metodo:

```
public int calculate(int i){
```



```
int k = i * 5;  
i = 2;  
return k * i;  
}  
  
public int calculate(int i){  
    Int k = i * 5;  
    Int j = 2;  
    return k * j;  
}
```



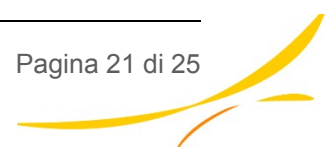
1 Regole e Suggerimenti

- ✓ Accedere ad ogni singola variabile di istanza attraverso i metodi getter e setter.
- ✓ Non inserire codice html (o in generale logica di presentazione) all'interno delle classi java.
- ✓ Non inserire codice java (o in generale logica business) all'interno di file jsp.
- ✓ Non inserire label fisse nelle jsp e utilizzare sempre il supporto di Internazionalizzazione:
 - utilizzare il tag `wp:i18n` apposito in elementi di Front-End
 - utilizzare il supporto di Struts2 per elementi di presentazione del Back-End
- ✓ Definire SEMPRE una interfaccia firma di ogni singolo bean (Spring Object). Referenziare il bean concreto sempre attraverso la sua interfaccia. Utilizzare questa procedura per:
 - ogni singolo jAPS Manager (classi che estendono `AbstractService`)
 - ogni singolo DAO (classi che estendono `AbstractDAO`)
 - ogni singola Action (Struts2 Object) (classi che estendono `BaseAction`)
- ✓ Come regola generale, NON nascondere le eccezioni che si potrebbero generale nei blocchi di codice java:
 - Nei jAPS Managers, tracciare adeguatamente e rilanciare le eccezioni sempre nella maniera con cui vengono tracciate le eccezioni nel jAPS Core.

Dai jAPS Manager non rilanciare mai eccezioni diverse da `ApsSystemException`

Esempio

```
try {  
    ...  
    ...  
}
```



```
} catch (Throwable t) {  
  
    ApsSystemUtils.logThrowable(t, this, "<METHOD_NAME>");  
  
    throw new ApsSystemException("<DESCRIPTION>", t);  
  
}
```

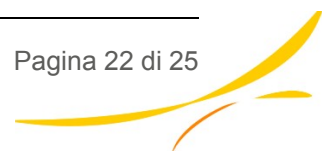
- Nelle classi Action, tracciare adeguatamente le eccezioni con il log applicativo e, dove richiesto un result, restituire FAILURE in caso di eccezione.

```
public String changePassword() {  
  
    try {  
  
        .....  
  
    } catch (Throwable t) {  
  
        ApsSystemUtils.logThrowable(t, this, "changePassword");  
  
        return FAILURE;  
  
    }  
  
    return SUCCESS;  
  
}
```

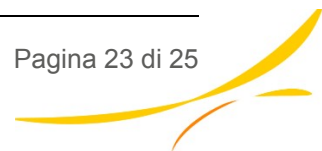
- ✓ Implementare adeguatamente il metodo init() dei jAPSManger, logando SEMPRE la inizializzazione avvenuta.

```
public void init() throws Exception {  
  
    .....  
  
    ApsSystemUtils.getLogger().config(this.getClass().getName() + "  
inizializzato ");  
  
}
```

Inserire eventuali info nel log (esempio: se il metodo di inizializzazione carica un insieme di elemento, tracciare il numero di elementi caricati)



- ✓ Nell'iniezione di jAPS Manager (o in generale dei Bean di Spring), utilizzare sempre la tecnica del Dependency Injection. Nei limiti del possibile, non estrarre i Manager (o Bean in generale) direttamente dall'ApplicationContext di Spring.



1 Errori comuni e suggerimenti

- ✓ Conviene sempre fare i metodi getters e setters delle variabili statiche (variabili di classe non final); i getters sono normalmente public mentre i setters sono di solito protected.
- ✓ Cercare di utilizzare la classe StringBuffer al posto della classe String (quando ci sono da fare concatenazioni) perché risulta essere migliore in termini di performance.
- ✓ Garantire la leggibilità del codice all'interno di ogni singolo metodo cercando di non annidare istruzioni all'interno di altre:

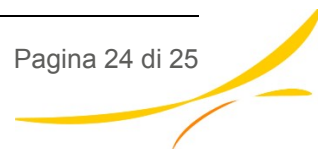
```
int var = this.extractValue(this.getOperator(this.getX()),  
this.getAdvisor(y, z));
```

```
int operator = this.getOperator(this.getX());
```

```
int advisor = this.getAdvisor(y, z);
```

```
int var = this.extractValue(operator, advisor);
```

- ✓ **ATTENZIONE** una classe final non consente sottoclassi.
- ✓ Per il nome dei package si usa il singolare e minuscolo.
- ✓ Le interfacce devono avere il prefisso "I" (IRunnable, Icustomer)
- ✓ Mantenere all'interno della classe ordine di variabili e campi:
 - costruttori
 - metodi pubblici
 - metodi privati
 - dichiarazione variabili di classe private
 - dichiarazione variabili di istanza private
 - dichiarazione variabili di classe pubbliche



- dichiarazione variabili di istanza pubbliche
 - dichiarazione variabili di istanza protected
 - dichiarazioni costanti di classe
- ✓ Non confondere l'operatore di assegnazione con quello di uguaglianza:

```
If (x == 0).....
```

```
If (x = 0)..... // qui il compilatore non rileva l'errore
```

```
If (0 == x).....
```

```
If (0 = x)..... //qui l'errore viene rilevato in fase di compilazione
```

