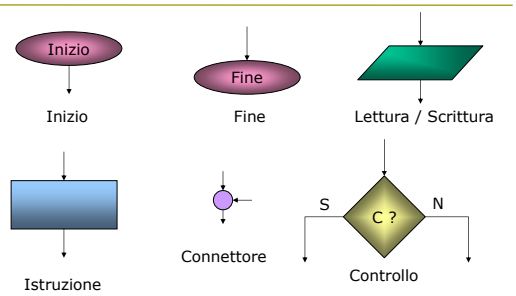
Flow Chart

Prof.ssa Gentile
a.a. 2011 - 2012

Flow Chart

- È un linguaggio formale di rappresentazione bidimensionale degli algoritmi che serve per evidenziare l'ordine di esecuzione delle istruzioni.

Blocchi elementari



Validità dei Flow Chart

- Descrive un algoritmo se:
  - Ha un blocco iniziale e uno finale
  - È costituito da un numero finito di blocchi istruzione e/o blocchi lettura/scrittura e/o blocchi controllo
  - Ciascun blocco elementare rispetta le condizioni di validità

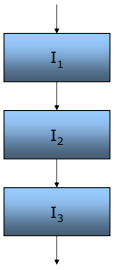
Condizioni di validità

- Ciascun blocco istruzione e blocco lettura/scrittura ha una sola freccia entrante e una solo freccia uscente
- Ciascuna freccia entra in un blocco o si innesta su un'altra freccia
- Ciascun blocco è raggiungibile dal blocco iniziale
- Il blocco finale è raggiungibile da qualsiasi altro blocco

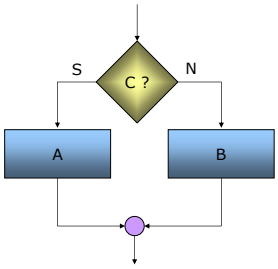
Analisi Strutturata

- Si definisce strutturata una combinazione di blocchi che rispetta i seguenti schemi di flusso:
  - Sequenza
  - Selezione
  - Iterazione

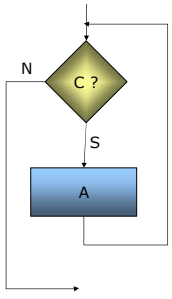
Sequenza

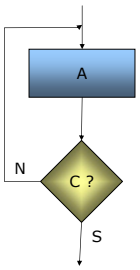


Selezione



Iterazione



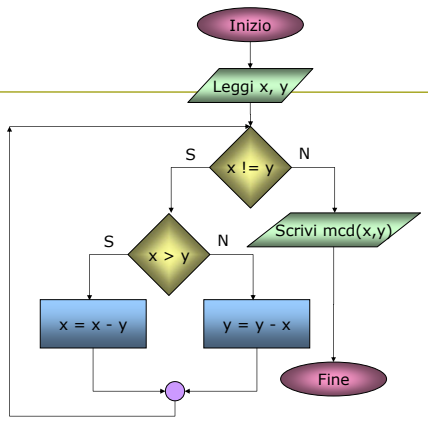


Ciclo

- Ciclo enumerativo
  - Quando è noto a priori il numero di volte che deve essere eseguito
- Ciclo indefinito
  - Quando non si conosce il numero di volte che deve essere eseguito

Teorema di Bohm - Jacopini

- Il teorema di Bohm-Jacopini afferma che un qualsiasi algoritmo può essere sempre costruito in modo equivalente utilizzando soltanto le tre strutture di programmazione:
  - sequenza
  - selezione
  - iterazione



Massimo Comune Divisore

```
import java.lang.*;

class MCDApp {

static int mcd(int x, int y) {

while ( x != y ) {
    if ( x > y ) {
          x = x - y;
    } else {
          y = y - x;
    }
}
return x;
}
```

```
// Inizio del programma.

public static void main(String args[]) {

int x;
int y;

x = Integer.valueOf(args[0]).intValue();
y = Integer.valueOf(args[1]).intValue();

System.out.println( "Il massimo comune
     divisore tra " + x + " e " + y + " è " +
     mcd( x, y ) );
 }
}
```


Linguaggio Java

Tipi primitivi Java

Tipo	Dimensione	Intervallo di valori
boolean	1 bit	true o false
char	16 bit	carattere unicode
byte	8 bit	[-128, 127]
short	16 bit	[-32768, 32767]
int	32 bit	[-2147483648, 2147483647]
long	64 bit	[-9223372036854775808, 9223372036854775807]
float	32 bit	[-3,4E38, 3,4E38]
double	64 bit	[-1,7E308, 1,7E308]

Parole Chiave

abstract	**default**	**goto**	**null**	**synchronized**
boolean	**do**	**if**	**package**	**this**
break	**double**	**implements**	**private**	**throw**
byte	**else**	**import**	**protected**	**throws**
case	**extends**	**instanceof**	**public**	**transient**
catch	**false**	**int**	**return**	**true**
char	**final**	**interface**	**short**	**try**
class	**finally**	**long**	**static**	**void**
const	**float**	**native**	**super**	**while**
continue	**for**	**new**	**switch**	**enum**

Sequenze di escape

\n	newline, va a capo
\t	tab, tabulazione
\b	backspace, cancella a sinistra
\r	return, rappresenta il carattere speciale INVIO
\f	from feed
\\	è il carattere '\', blackslash
\'	apice
\"	virgolette
\d	carattere rappresentato in base ottale
\xd	carattere rappresentato in base esadecimale
\ud	carattere rappresentato unicode

Operatori

- Aritmetici (+, -, /, *, %)
- di Confronto (==, !=, >, <, >=, <=)
- sui Bit (>>, <<)
- di Assegnazione (=, +=, -=, /=, …)
- di Concatenamento (+ tra stringhe)
- di Condizionamento (?:)
- di Conversione (casting)
- Logici (&&, ||, !, &. |, ^, ~)

Operatori

- Unari:
  - ++x ; x++ ; y-- ; --y ;
- Binari:
  - b + c; a = 5;
- Ternari
  - espressione_test ? azione_true : azione_false;

Esempio operatore ternario

```
int a = 5;
int sign = a > 0 ? 1 : -1;
```

```
int sing = a > 0 ? 1 : (a == 0 ? 0 : -1);
```

```
int sign;
if (a > 0)
   sign = 1;
else if (a == 0)
   sign = 0;
else
   sign = -1;
```


Dichiarazione e Assegnazioni

- int a; //dichiarazione
- a = 5; // assegnazione
- float b=3.6f; //dic. e ass.
- new Auto(); //istanza della classe
- Auto vettura = new Auto(); //dic. e ass. vettura

Flussi di Controllo

- Istruzioni condizionali:
  - if else
  - switch case default
- Istruzioni iterative
  - while
  - do while
  - for
  - for each

if else

```
if (condizione) {
  // blocco istruzioni1;
}
else {
  // blocco istruzioni2;
}
```


switch case default

```
switch (<espressione intera>) {
case (<valore costante 1>):
//… (<sequenza di istruzioni)
break;
case (<valore costante 2>):
//… (<sequenza di istruzioni)
break;
//…
default:
// è opzionale
//… (<sequenza di istruzioni>)
}
```

Istruzioni iterative

```
while (condizioni di permananza) {
  // blocco istruzioni;
}

do {
  // blocco istruzioni;
} while (condizione di permanenza);
```


Costrutto for

```
for (int i=0; i<10; i++) {
    // blocco di istruzioni;
}
```

Costrutto for each

```
for (variabile_temporanea : oggetto_iterabile)
{
    // blocco di istruzioni;
}

public int sum(int[] a) {
    int result = 0;
    for (int i : a)
          result += i;
    return result;
}
```


Array

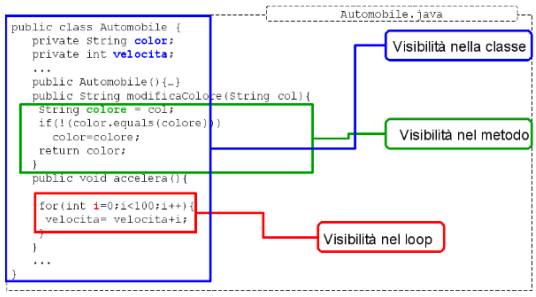
```
int [] arrayOfInt = {10, 20, 30, 40, 50, 60};
// è equivalente a
int [] arrayOfInt = new int[6];
arrayOfInt [0] = 10; arrayOfInt [1] = 20; arrayOfInt [2] =
30; arrayOfInt [3] = 40; arrayOfInt [4] = 50; arrayOfInt [5]
= 60;


String [][] names = {{"Sig. ", "Dott. ", "Ing. "},
                     {"Rossi", "Verdi"}};
System.out.println(names[0][0] + names[1][0]); // Sig. Rossi
System.out.println(names[0][2] + names[1][1]); //Ing. Verdi
```

Scope delle variabili

- Lo scope di una variabile è la visibilità di una variabile, cioè la parte di programma in cui quella variabile può essere referenziata, inizia con la dichiarazione e finisce con la chiusura del blocco
- Le variabili possono essere dichiarate a livello di:
  - loop: visibilità solo nel ciclo;
  - metodo: visibilità in tutto il corpo del metodo;
  - classe : visibilità in qualunque parte del codice.

Esempio



Metodo main()

```
public static void main(String[] args)
   // Definiamo il metodo Main.
   {                          // Inizio del main()
     …
     }          // Fine del metodo principale Main()
```

Public = accessibile da tutti
Static = unico
Void = non restituisce alcun valore al ritorno
String [] args = serve per salvare gli eventuali parametri passati al programma

```
import java.io. ;
// Include le funzioni I/O standard di base.

class Spippo {
// Inizia la classe Spippo

public static void main(String[] args)
//  Definiamo il metodo Main. Il programma inizia
    l'esecuzione da qui.
{
 // Inizio del main()

System.out.print("Pippo\n");
//  Chiamo la funzione print del package java.io. Il \n indica
   di andare a capo.
}            // Fine del metodo principale Main()
}          // Fine della classe Spippo
```


Lezione di stile in Java

In generale

Nomi descrittivi

- Utilizzare nomi che descrivano lo scopo dell'elemento. Ad esempio, una classe che accede ad un database per leggere informazioni sul cliente:
  RecuperoDati [NO]
  CaricatoreDatiCliente [Si]
- Oppure, in caso di un metodo che ritorna la descrizione di un oggetto
  public String ritornaStringa(); [No]
  public String getDescrizione(); [Si]

Nomi pronunciabili

- Utilizzare nomi facilmente pronunciabili, e di uso comune, che non abbiano doppie in posti critici.

Abbreviazioni

- Non utilizzare abbreviazioni, ma parole complete. Ad esempio, negli attributi:
  float valMedCamp; [No]
  float valoreMedioCampione; [Si]

Package

- Il nome del package dovrebbe essere univoco, scritto in tutte lettere minuscole e dovrebbe iniziare da domini di livello più alto (come gov, com, it). Ogni componente del nome è separato da un punto (.).
- Il nome dovrebbe includere il proprio dominio Internet, in modo da assicurarne l'univocità a livello di azienda.

  it.bigatti.iSafe

  com.sun.internal

Classe

- Una classe è contenuta all'interno di un file con estensione .java e segue questa struttura:
  - commento con l'intestazione del file;
  - dichiarazione del package;
  - sezione import;
    - package interni;
    - package estensioni;
    - package libreria di terze parti;
    - package applicativi;
  - dichiarazione della classe;
    - attributi pubblici;
    - attributi privati;
    - attributi protetti;
    - altri attributi;
    - costruttori;
    - altri metodi;
  - classi interne;
- Nel file è obbligatoria solo la definizione della classe; il resto degli elementi è opzionale. Ciascun elemento è separato dagli altri da una riga vuota, come ciascun metodo

Esempio

```
class Cliente extends AbstractElemento
        implements Serializable {

        public static Cliente _instance;
        int id;
        //…

        public Cliente() {
        }

        public int getID() {
                return id;
        }

        public String getRagioneSociale() {
                return ragioneSociale;
        }
}
```

Sezione import

- La sezione relativa all'importazione dei package e delle classi dovrebbe seguire queste regole:
  - **ordine alfabetico**. I package dovrebbero essere inseriti in ordine alfabetico;
  - **raggruppati**. I package dovrebbero essere raggruppati come segue:
    - package della libreria di base (java.*);
    - package delle estensioni standard (javax.*);
    - package delle librerie aggiuntive (p.e. org.apache.*, org.w3c.dom.*);
    - package dell'applicazione che si sta sviluppando;
  - **separati**. Tra un gruppo e l'altro deve essere presente una riga vuota;

Esempio

```
import java.io.*;
import java.sql.*;
import java.text.*;
import java.util.*;

import java.xml.parsers.*;
import javax.xml.transform.*;

import org.s3c.dom.*;
import org.xml.sax.*;
import org.xml.sax.ext.*;

import it.bigatti.iSafe.*;
import it.bigatti.util.*;
```


Dichiarazione della classe

- La dichiarazione della classe dovrebbe essere disposta su più righe, in modo da limitarne l'ampiezza. Una buona soluzione è spezzare la riga prima della parola chiave *implements*:

```
class ElencoProdotti extends AbstractElenco
                implements Storeable, Cacheable
```

Nomi

- Il nome di una classe è composto da una serie di sostantivi concatenati, la cui iniziale è scritta in maiuscolo.
- Ad esempio:
  - Cliente
  - ElencoClienti

Uniformità della lingua e dei termini

- **argomenti verticali**. Lo sviluppatore medio non conosce il vocabolario specifico di alcuni segmenti applicativi;
- **complessità eccessiva**. Chi crea una classe, con un nome esotico, deve poi condividere questa parte del programma con il resto del gruppo di lavoro;
- **disomogeneità**. Se nell'interfaccia si parla di cliente, mentre nel dominio si trova un Customer, la leggibilità del sistema nella sua integrità è ridotta;
- **rapporto con il cliente**. Si immagini un intero progetto in cui si fa riferimento agli stessi concetti con due insiemi di termini differenti.

Soluzione possibile

- Utilizzare termini in italiano per le classi funzionali e in inglese per quelle più a basso livello.
- Le classi funzionali sono quelle classi relative al dominio dell'applicazione, come Cliente, Fornitore, Ordine;
- Le classi tecniche a livello inferiore possono riguardare il caching, l'accesso al database o all'infrastruttura Web (servlet/JSP).

Nominare i package

- Il nome della classe non deve essere troppo lungo. Il package di appartenenza deve fornire il contesto utile per ridurne il nome.
- I nomi troppo lunghi riducono la leggibilità del codice, e sono comunque indice di un non ottimale utilizzo del linguaggio, in quanto i package dovrebbero definire un contesto entro cui agire.
- I nomi delle classi dovrebbero dare solo l'elemento distintivo finale.

Parti standard del nome

- È possibile indicare che una determinata implementazione è quella di default utilizzando il termine "Default".
  - DefaultGruppo
  - DefaultUtente
- Le classi astratte iniziano sempre per Abstract:
  - AbstractCliente
  - AbstractProdotto
- Le classi che utilizzano il pattern model-view-controller devono terminare con un postfisso standard, che identifica la tipologia del componente. Ad esempio:
  - ListModel
  - TreeModel
  - PlainView
  - DefaultListController
- Le eccezioni terminano invece sempre per Exception:
  - UtenteNonTrovatoException
  - GiacenzaInsufficienteException

Classi di elenco

- Le classi che rappresentano un elenco di elementi devono avere un elemento distintivo. Una possibilità è utilizzare il plurale:
  - Clienti (elenco clienti)
  - Prodotti (elenco prodotti)
- Questo approccio però tende a creare classi dal nome troppo simile a quello della classe di cui contengono l'elenco, dove cambia una sola lettera. Una soluzione migliore è l'utilizzo del termine "Lista" o "Elenco".
  - ElencoClienti
  - ListaProdotti

Metodi

- Per quanto riguarda i metodi è bene utilizzare metodi in italiano per le classi funzionali in inglese per quelle tecniche:
  - elaboraRisultato();
  - caricaDati();
  - eseguiTransazione();
- Sono sufficientemente leggibili, anche se generici (il contesto dovrebbe essere dato dalla classe in cui sono inseriti).
- I metodi dovrebbero essere verbi, dove ciascuna parola che compone il nome ha l'iniziale in maiuscolo, tranne la prima.

Convenzioni Javabeans

- Prevedono l'utilizzo di prefissi per indicare lo scopo di determinati metodi:
  - **getXXX().** Definisce un metodo che ottiene il valore di una proprietà (getter);
  - **setXXX().** Definisce un metodo che imposta il valore di una proprietà (setter);
- Nel caso che il valore sia di tipo boolean il getter usa il prefisso "is" oppure "has"
  - **isWritale()**
  - **hasData()**

Esempi

Inglese
- getCustomer()
- getID()
- setTimestamp()
- setName()
- isWritable()
- isUpdatable()
- isAvailable()
- hasChild()
- hasChild()

Italiano
- ritornaCliente()
- ritornaID()
- impostaMomento()
- impostaNome()
- èScrivibile()
- èAggiornabile()
- èDisponibile()
- haFigli()
- possiedeFigli()

Problemi

- **accentate**. L'italiano ci costringerebbe ad utilizzare caratteri accentati come "è". Sebbene Java sia un linguaggio UNICODE, spesso si decide di "non fidarsi", e di compilare i file sorgenti in ASCII standard;
- **difformità**. Il codice realizzato è difforme dal codice Java esistente, come le librerie di base di Java o la parte più tecnica dell'applicazione, rendendo più difficoltosa le leggibilità del codice;
- **incompatibilità**. Le convenzione di codifica dei Javabeans sono state sviluppate per rendere utilizzabile un componente all'interno di strumenti di sviluppo visuali.

Soluzione: Usare i prefissi in Inglese

- In alternativa è possibile lasciare in inglese i prefissi, utilizzando l'italiano solo per la parte principale del nome:
  - getCliente()
  - getID()
  - setTimestamp()
  - setNome()
  - isScrivibile()
  - isAggiornabile()
  - isDisponibile()
  - hasFigli()

Dichiarazione del metodo

- In modo similare alle classi, i metodi vengono dichiarati anche su due righe, spezzando la prima riga prima della parola chiave throws.

```
void loadProdotto (String gruppo, String id)
      throws ProdottoNonTrovatoException;
```

Parametri

- I parametri di un metodo dovrebbero rispettare una serie di convenzioni:
  - **nomi descrittivi**. Il nome del metodo dovrebbe dare il senso del significato del parametro, come: codiceProdotto, tipoOperazione, ordineDaInoltrare;
  - **variabili generiche**. In caso di variabili generiche, il nome della variabile è uguale a quello della classe, tranne per l'iniziale in minuscolo.

    aggiungi(Cliente cliente)
    addMessaggio(Utente utente, Messaggio messaggio)

Valore di ritorno

- Il valore da ritornare dovrebbe essere contenuto in una variabile, dichiarata ad inizio metodo e ritornata alla fine dello stesso. Ad esempio:

```
float getPrezzo() {
    float prezzo = 0.0;
      prezzo = costo – (costo * sconto) / 100;
      prezzo += prezzo * TabellaIVA.getValore( codiceIVA );
      //… altre elaborazioni
      return prezzo;
 }
```

- Non devono esistere altre istruzioni return a parte quella a fine metodo. Nel caso ne vengano utilizzati diversi, può non apparire chiaro, in una determinata esecuzione, quale di queste istruzioni venga invocata e di conseguenza risulta più complesso capire il codice.

Campi e variabili

- I **nomi** dei campi sono costruiti come i nomi dei metodi, concatenando nomi ed aggettivi, scrivendo in maiuscolo l'iniziale a partire dalla seconda parola.

```
String descrizione;
int id;
String codiceProdotto;

List elencoProdotti;
```

Costanti

- Le costanti sono attributi marcati come final e che hanno nomi completamente in maiuscolo, eventualmente composti da più parole.
- Le singole parole sono separate da underscore (_). E' buona norma specificare almeno due termini, il primo definisce il contesto della costante. Ad esempio:
  - COLORE_ROSSO
  - PRIORITA_ALTA
- Se esiste una sola tipologia di costanti all'interno di una specifica classe, è possibile omettere il prefisso, in quanto in contesto è già dato dalla classe:
  - Colore.ROSSO
  - Livello.MEDIO

Algoritmi

Dimensioni

- La dimensione di un singolo algoritmo (o metodo) non dovrebbe eccedere la pagina, in modo che sia leggibile in una schermata e non richieda scorrimenti verticali. Dove l'algoritmo sia troppo complesso, eseguire un refactoring per separarlo in diversi sotto-metodi più semplici.

Blocchi di codice

- Le parentesi graffe devono iniziare sulla riga relativa all'istruzione e terminare sulla colonna di inizio dell'istruzione stessa. Ad esempio:

```
if ( a == 5 ) {
        //…
}

while ( true ) {
        //…
}

do {
        //…
} while( !esci );
```

- L'indentazione delle espressioni if dovrebbe essere di 8 spazi, in modo da staccare chiaramente dal codice all'interno delle graffe:

```
if( (prodotto.prezzo > Prodotto.prezzoMinimo ) &&
        prodotto.giacenza() != 0 ) {
        //…
}
```

Espressioni

- Nelle espressioni, valutare la posizione di termine della riga, per spezzarla nel punto più appropriato, ad esempio in corrispondenza di parentesi:
  ```
  float prezzoFinale = (prezzo – (prezzo * sconto) / 100);
  ```
- Per quanto riguarda l'operatore ternario, esistono tre possibilità, la cui scelta dipende molto dalla lunghezza di beta e gamma:
  ```
  alpha = (aLongBooleanExpression) ? beta : gamma;

  alpha = (aLongBooleanExpression) ? beta
                                   : gamma;

  alpha = (aLongBooleanExpression)
      ? beta
      : gamma;
  ```
- Ogni riga dovrebbe contenere al massimo una espressione:
  ```
  indiceProdotto++; elabora(); [No]

  indiceProdotto++;
  elabora();
  ```


Condizioni

- L'istruzione if deve rispettare le seguenti regole:
  - **sempre parentesi**. Utilizzare sempre le parentesi graffe, anche se segue solo una istruzione, allo scopo di ridurre eventuali errori di codifica nel momento che si desiderino aggiungere ulteriori istruzioni condizionate alla if. Inoltre, il blocco di codice risulta più leggibile nel momento che lo sviluppatore si abitua ad utilizzare questo standard.

Iterazioni

- I cicli devono seguire le seguenti regole:
  - **contatore interno**. Dichiarare il contatore direttamente nel ciclo, in quando si riduce l'area di attenzione che lo sviluppatore deve porre a questo aspetto del programma. Ad esempio: for(int ii = 0; ii < 5; ii++);
  - **contatori "cercabili"**. Utilizzare i classici contatori "i" (derivati dalla prassi utilizzata dai matematici) per i cicli, complica le ricerche nel codice, in quanto la ricerca del carattere "i", ritorna sempre un numero elevato di occorrenze (vengono individuate anche le i delle istruzioni if, ad esempio). Per contro, contatori come indiceCiclo, sono troppo lunghi; una soluzione semplice è utilizzare contatori a due caratteri, come ii, kk, jj. Questa convenzione può essere applicata anche solo in classi di grossa dimensione;

Variabili

- Le variabili dei metodi (ma le stesse regole si applicano alle variabili di instanza e di classe) devono rispettare le seguenti linee guida:
  - **visibilità**. Utilizzare l'area di visibilità più stretta possibile, dichiarando le variabili appena prima del loro utilizzo. Questo consente di limitare il raggio di attenzione che lo sviluppatore deve porre.
  - **semantica singola**. Una variabile non deve servire a due scopi, sebbene sia più compatto utilizzare una generica variabile contatore per due cicli diversi, è più leggibile dichiararne diverse in modo da specificare un nome più descrittivo. Ad esempio:

```
void controlla() {
    int numeroControlliFormali = 0;
    while( controlliFormali( numeroControlliFormali ) ) {
            numeroControlliFormali++;
    }
    setNumeroControlliFormali( numeroControlliFormali );

    int numeroControlliAggiuntivi = 0;
    while( controlliAggiuntivi( numeroControlliAggiuntivi ) ) {
            numeroControlliAggiuntivi++;
    }
    setNumeroControlliAggiuntivi( numeroControlliAggiuntivi );
}
```


Chiamata a metodi

- Non utilizzare spazi dopo il nome del metodo. Ad esempio:
  elabora (5,prodotto, "prova"); [No]
  elabora(5, prodotto, "prova");
- Se l'elenco dei parametri è lungo, è possibile spezzarlo per renderlo più chiaro:

```
prodotto.aggiorna( gruppo, descrizione, costo, quantita,
    disponibilta, immagine );
```

- o in alternativa:

```
prodotto.aggiorna(
gruppo,
descrizione,
costo,
quantita,
disponibilta,
immagine
);
```


- La seconda versione è utile quando le variabili da passare siano lunghe oppure che il valore sia il risultato di una chiamata più complessa:

```
prodotto.aggiorna(
    gruppo,
    altroProdotto.getDescrizione(),
    Catalogo.getCatalogoDefault().getCosto( prodotto.id ),
    quantita,
    disponibilta,
    CommonUtils.loadImmagine( prodotto.id );
);
```

Documentazione

- La documentazione nei programmi Java è redatta secondo lo standard Javadoc, che prevede la compilazione della documentazione direttamente all'interno del codice.
- In seguito, tramite il comando javadoc, questi vengono estratti e utilizzati per compilare la documentazione di progetto, solitamente in formato HTML.
- Per mantenere un approccio agile allo sviluppo, è necessario trovare un bilanciamento tra la necessità di documentare, e quella di concentrarsi sullo sviluppo.
- Due sono le ulteriori forze in gioco: da una parte il codice deve essere utilizzato da terzi, sia per la manutenzione che per un eventuale riutilizzo - e quindi ha la necessità di essere documentato – ma d'altra parte, la documentazione non deve offuscare il codice: la documentazione migliore è il codice ben scritto.

Convenzioni

- **non documentare**. Sembra un suggerimento assurdo ed al di fuori di ogni corrente di pensiero. Ma è consigliabile non documentare subito le classi che vengono sviluppate, in quanto non è certo che la classe appena realizzata resterà all'interno dell'applicazione ed in quella forma. Un approccio agile è dunque quello di non documentare subito tutto, ma aspettare che una determinata porzione di codice sia consolidata.

Convenzioni

- **codice auto-documentante**. Il codice dovrebbe essere sempre il più possibile chiaro, in modo che il suo funzionamento sia ovvio senza la necessità di scrivere commenti;
- **codice complesso**. Codice particolarmente complesso dovrebbe essere riscritto e non commentato, tramite operazioni di refactoring. Ovviamente ciò non è sempre possibile, e dunque brevi commenti esplicativi sono indispensabili;
- **commento mirato**. Nei casi di codice particolarmente complesso e non ulteriormente semplificabile, oppure per quelle informazioni che lo sviluppatore ritiene di dover associare al codice, utilizzare un commento per descrivere l'informazione;
- **non offuscare**. La documentazione dovrebbe essere inserita in modo da non rendere più complessa la lettura del codice.

Esempio

- Sbagliato:

```
if( a == 5 ) {
    /* commento */
    break;
}
```

- Corretto:

```
/* commento */
if( a == 5 ) {
    break;
}
```


Convenzioni

- **non commentare il linguaggio**. I commenti devono riportare informazioni sul perché viene eseguita una determinata operazione e non sulla sintassi del codice.

```
i++; //incrementa i     [No]
i++; //elabora il prossimo cliente    [Si]
```

- **italiano**. Nelle parti descrittive sarebbe buona norma utilizzare tutti termini italiani. Termini di uso comune inglesi, come file o directory possono essere mantenuti, mentre il plurale non utilizza la "s" finale (no files ma file);
- **commenti cercabili**. Se una determinata porzione di codice richiede un intervento successivo, sia di ottimizzazione che di correzione, marcarlo con un commento opportuno, con un prefisso standard:
  - //TODO: descrizione.

Dove usare i commenti

- Se è necessario commentare una sola riga, utilizzare il commento a singola linea (//), sia che questo sia posizionato alla riga precedente (soluzione migliore), che a fine riga:

```
//elaboro il prodotto successivo
indiceProdotto++;

indiceProdotto++; //elaboro il prodotto successivo
```

- Nel caso si debba commentare un blocco di codice, utilizzare /* … */.

Commenti

- I commenti /** … */ sono riservati alla documentazione Javadoc.
- Evitare di costruire riquadri grafici, in quanto richiedono del tempo per essere costruiti e mantenuti allineati quando se ne modifica il contenuto:

```
/*******************
* Classe Cliente *
* *
* Implementa la *
* gestione di un *
* cliente *
*******************/
```


Esercizi

- Somma tra due numeri positivi
- Moltiplicazione di due numeri positivi attraverso la somma
- Divisione intera tra due numeri positivi
- Elevamento a potenza
- Radice quadrata
- Fattoriale
- Massimo comune divisore
- Numero primo
- Somma dei primi 100 numeri naturali

Somma dei primi 100 numeri naturali

```
public class CicloFor {
  public static void mail (string [] args){
     int somma=0, i;
     for (int i=0; i<100; i++)
          somma ++;
     system.out.println ("la somma è =" +
  somma)
  }
}
```

Esercizi sugli array

- Ricerca sequenziale
- Ricerca binaria
- Bubblesort
- Torre di Hanoi
- Quicksort
- Permutazioni