OOPs! Programmo...

Indice

Indice

Premessa per noi poveri insegnanti che dobbiamo convertirci da imperativo a OOP

Typically an Object Oriented (OO) application consists of a network of objects working together to accomplish a goal.
(http://www.theserverside.com/tt/articles/article.tss?l=JMockTestDrivenDev)

Paradigma imperativo

La programmazione imperativa si basa sul concetto di algoritmo.
Sviluppare una applicazione vuol dire realizzare un programma che implementa un algoritmo risolutivo che manipola i dati seguendo adeguate successioni di passi.

Pertanto il percorso di apprendimento della tecnica imperativa parte dal concetto dato e dalla capacità di memorizzarlo/elaborarlo.
Conseguenza di ciò è lo studio di variabile e istruzione, con l'obiettivo di far comprendere come rappresentare dati e come manipolarli.
Poi si prosegue articolando sempre più complessità sui dati e sugli algoritmi.
Quando la complessità supera una certa soglia si passa alla scomposizione procedurale/funzionale.

Paradigma OOP

La programmazione OOP si basa sul concetto di oggetto.
Sviluppare una applicazione vuol dire realizzare un programma che utilizza una serie di oggetti dotati di capacità elaborative che portano alla soluzione del problema.

Per OOP il punto nodale sono gli oggetti. Il primo concetto da acquisire è di saper utilizzare oggetti che hanno la capacità di fare/gestire ciò che serve, o di produrne di nuovi con questo obiettivo.
Quindi usare oggetti, definire una classe e istanziare oggetti. Definire metodi.
La capacità di memorizzare informazione è detenuta da un oggetto che offre anche i relativi metodi di manipolazione.
Aumentare complessità significa aumentare il numero di oggetti e/o aumentare il numero di informazioni interne agli oggetti e aumentare complessità elaborative dei metodi.
All'interno dei metodi incontriamo i tre costrutti base di Jacopini Bohm.
E ancora la capacità degli oggetti di dialogare tra di loro.
E la capacità delle classi di strutturarsi fra di loro.

Quindi...

Cerco qui di costruire il percorso di insegnamento/apprendimento che si fondi sula scaletta OOP eventualmente attingendo alle consuetudini consolidate per l'imperativo solo dove risultano essere utili.

Il mondo ad oggetti

Nella nostra esperienza quotidiana facciamo uso continuativo di oggetti che utilizziamo per raggiungere i nostri scopi.

Ad esempio siamo abituati a prendere in mano il cellulare e ottenere la memorizzazione informazioni, comunicare ecc ecc
Quanto ci serve è sapere cosa è in grado gestire, operare e quali sono i comandi, le azioni da operare su di lui per raggiungere l'obiettivo.

D'altra parte sappiamo anche comporre oggetti diversi per ottenere funzioni che individualmente non sono in grado di svolgere, ad esempio con l'accoppiata lettore dvd-televisore si riesce a godere di un buon film su DVD, cosa non possibile se si usasse uno solo dei due oggetti.

Possiamo anche indurre un ragionamento a ritroso: quello che noi noi vediamo ed usiamo come un oggetto unico a sua volta "all'interno" è costituito di più oggetti, ognuno dei quali svolge compiutamente la sua parte.

Problemi ed oggetti

Nel nostro agire quotidiano ci è naturale affrontare e gestire le nostre esigenze individuando di volta in volta l'oggetto che ha le caratteristiche idonee a realizzare il nostro obiettivo.
Se devo scrivere su carta cerco una biro.
E anche desidero tanto il nuovo modello di lettore MP3 perché ha molti più GB di memoria per portarmi sempre dietro tutta la mia collezione di musica!

Oggetti, classi e istanze

Abbiamo dunque introdotto l'importanza che gli oggetti rivestono nel nostro quotidiano.
Completa il quadro l'osservare che ogni oggetto è la realizzazione concreta di una categoria di oggetti tutti dalle medesime caratteristiche.
Chiamiamo **classe** una categoria che definisce le caratteristiche di un certo insieme di oggetti.
Gli oggetti di un categoria possono essere assolutamente identici (ad esempio più biro identiche) oppure pur avendo le stesse caratteristiche differiscono per un particolare "valore" come ad esempio il colore dell'inchiostro: una biro rossa e una biro blu.
Il legame fra oggetto e classe viene chiamato **istanza**. Il termine non è molto diffuso nel linguaggio corrente, ma occorre farci abitudine...
Si dice che un oggetto è una istanza di una classe.

L'informatica, i programmi, gli oggetti

Nel mondo informatico abbiamo due grandi aree.
Una è costituita dalle parti materiali (hardware) i cui oggetti si vedono e toccano, come ad esempio il monitor, il mouse, il router, il cavo, eccetera.
L'altra non tangibile (software) i cui oggetti sono chiamati programmi, icone, messaggi.
Ci interessiamo di quest'ultima area che è oggetto di studio del corso.
Questi oggetti ci permettono di svolgere attività anche molto diverse: navigare per siti web, giocare, comunicare in voce e video, scrivere un testo, modificare un'immagine...

Ancora una volta sono oggetti che utilizziamo per raggiungere un nostro scopo.
Abbiamo a disposizione stimoli con i quali pilotiamo il comportamento desiderato, il click col mouse, i tasti o altri dispositivi come un joystick, un telecomando, eccetera.

Lo sviluppo del software

Come si producono questi oggetti software?
L'insieme di tecnologie e metodologie di lavoro che portano alla realizzazione di questi oggetti viene chiamata "sviluppo" del software, ed è sostanzialmente la struttura portante di questo corso.
L'enfasi data agli oggetti in questi paragrafi introduttivi serve a introdurre la metodologia che attualmente risulta essere più efficace per lo sviluppo. Secondo questa metodologia si giunge al prodotto finito realizzando uno o più oggetti che composti fra loro realizzano l'obiettivo.
O più precisamente si progettano una o più classi con cui istanziare oggetti che permettono di raggiungere l'obiettivo.

Agire con gli oggetti

Azione

Nella nostra pratica quotidiana raggiungere un obiettivo significa svolgere una o più azioni che appunto realizzano lo scopo.

Obiettivi semplici si realizzano con una sola azione, ad es. Il lancio della palla da un giocatore ad un altro, o accendere la luce in una stanza.

Un obiettivo complesso richiede una successione di azioni, ad esempio prepararsi un piatto di pasta richiede scaldare all'ebollizione l'acqua, buttare giù la pasta, attendere la cottura, scolarla e infine condirla.

Esecutore

L'azione richiede la presenza di un soggetto in grado di svolgerla: l'esecutore.

Caratteristica fondamentale per la buona riuscita dell'azione è che l'esecutore abbia le capacità (abilità) necessarie per l'azione stessa.

Ripensiamo all'azione di accendere la luce: è piuttosto 'semplice', basta premere l'interruttore. Direi che più o meno tutti sono in grado di farlo. Ma non se l'interruttore è a un metro e mezzo di altezza e tu sei un bambino alto un metro, o sei immobilizzato su un letto...

In generale ogni azione richiede alcune capacità necessarie per essere eseguita.

Vista dall'altro lato ogni esecutore ha delle capacità che individuano un insieme di azioni che è in grado di eseguire

Istruire

Abbiamo più volte detto che raggiungiamo l'obbiettivo eseguendo una o più azioni. Ma perché "quella" (o quelle) azione?

Perché "sappiamo" che ci permette di raggiungere lo scopo.

Molto spesso otre a sapere cosa serve per raggiungere lo scopo siamo in grado di farlo.

Ad es.: sono al buio nella stanza, decido di accendere la luce, vado all'interruttore e lo premo accendendo la lampadina.

Ma se mi trovo a letto con una gamba ingessata il sapere cosa fare non basta... allora chiamo un'altra persona e chiedo di accendere la luce, ed ecco che la lampada è accesa.

In quest'ultimo caso ecco distinte le due funzioni: chi sa come raggiungere un obiettivo e chi è in grado di farlo ed agisce.

Da un lato di fronte al problema si è individuata la soluzione e comandata (o richiesta gentilmente...) l'attuazione all'esecutore.

Dall'altro lato l'esecutore, in grado di fare questo e molto altro, ha compiuto esattamente quell'azione perché sollecitato a farlo.

Programmare

Portandoci al mondo informatico il computer è un esecutore che ha una serie di capacità.

Lo sviluppo del software consiste nel predisporre la serie di istruzioni che sono finalizzate al raggiungimento dell'obiettivo.

Linguaggio

Chiarito il ruolo di progettista e di esecutore c'è un altro elemento cruciale.

Il passaggio da progettista ad esecutore è una comunicazione che deve essere chiaramente espressa dall'uno e ben compresa dall'altro.

A questo scopo deve essere utilizzato un linguaggio comune tra i due.

Tornando all'esempio della luce nella stanza il linguaggio potrebbe essere l'italiano, ma se l'unica persona in casa parla solo russo ci si potrebbe trovare in difficoltà...

Allora si cambia strategia, magari entrambi conosciamo l'inglese.

Quindi prima caratteristica è che il linguaggio deve essere comune.

Ma ricordo che non basta. Bisogna che vengano richieste azioni eseguibili. Se chiedo ad un bimbo di 10 anni 'mi risolvi questa equazione di secondo grado?' magari comprende ogni singola parola, ma non ha idea di come farlo...

Nel mondo informatico esistono dei linguaggi, detti di programmazione, che svolgono il ruolo di mediazione della comunicazione tra lo sviluppatore, progettista del software, e il computer, l'esecutore.

Caratteristiche di questi linguaggi è di essere comprensibili, cioè eseguibili, dal computer a patto che i comandi siano ben espressi.

E dal nostro punto di vista possiamo addestrarci a conoscerli ed esprimere secondo le regole proprie del linguaggio le nostre idee.

Regole di scrittura e significato

Una caratteristica rilevante dei linguaggi di programmazione è di essere **linguaggi formali** in cui cioè le regole di forma (regole di scrittura) sono ben definite e da applicare rigidamente. Siamo abituati ai linguaggi naturali con cui ci esprimiamo solitamente (italiano, inglese, ecc ecc) per i quali le regole esistono, ma con una maggiore flessibilità; siamo cioè in grado di interpretare correttamente anche frasi che contengono errori non troppo gravi.

I linguaggi di programmazione (come tutti i linguaggi) hanno delle regole che possono essere raggruppate in due aree.

Una prima area è quella delle regole di scrittura, della sintassi. Queste regole prevedono specifiche parole utilizzabili e la loro esatta scrittura, ad es in italiano 'casa' è corretta, ma non 'csa'. E ancora anche successioni come "l'aggettivo segue il sostantivo a cui si riferisce", es 'casa bianca'.

Una seconda area è quella del significato da attribuire alle frasi, la semantica. Normalmente una frase corretta sintatticamente produce anche un significato, ad es: "il bambino mangia la mela". Ma la frase "la mela mangia il bambino" è sintatticamente corretta (parole ben scritte, successione corretta soggetto verbo complemento) ma ha un

significato improbabile...

Nello sviluppo di software è normale incorrere in errori sia di sintassi che di semantica. E' necessario individuarli e correggerli. Di solito per gli errori sintattici il computer stesso ci viene in aiuto. Le regole sintattiche sono molto 'rigide' e il computer stesso segnala dove qualcosa 'non torna'.

Per la semantica invece dobbiamo noi adottare strategie che consentano di verificare. Infatti se il computer riceve istruzioni sintatticamente corrette ma non adeguate semanticamente fa delle cose diverse rispetto a quello che ci aspettiamo.

Quindi la verifica consiste nel sollecitare azioni di cui prevediamo l'esito e verificare l'esatta corrispondenza dei risultati.

Linguaggi di programmazione

Esistono molti diversi linguaggi di programmazione. Sono stati pensati in base a diverse metodologie di lavoro e/ in riferimento a specifici campi di applicazione.

Nel nostro contesto ci occupiamo di linguaggi di uso generale (general purpose). Fra i linguaggi cito per esempio Java, C++, C#, Pascal, PHP, Fortran, Cobol, Basic, Ruby, Pyton... l'elenco può essere lunghissimo.

In questo corso faremo riferimento al linguaggo Java, che fa esplicito riferimento alla programmazione orientata ad oggetti.

Il codice sorgente

Il primo passo per realizzare un programma è di scrivere il "**sorgente**" (in inglese source). E' in sostanza il testo del programma, scritto seguendo le regole proprie del linguaggio utilizzando i normali caratteri alfabetici, numerici, di punteggiatura e quindi in qualche modo leggibili anche da noi.

Il programma così preparato non è ancora nella forma in cui il computer è in grado di eseguire, occorrono alcuni passaggi. Il tipo di passaggi può essere diverso tra un linguaggio e l'altro secondo tre grandi categorie:

- linguaggi compilati
- linguaggi interpretati
- linguaggi a codice intermedio

Linguaggi compilati

Caratteristica propria dei linguaggi di questo tipo è che il sorgente viene fatto elaborare da un particolare programma chiamato **compilatore** che svolge due funzioni: verifica la correttezza sintattica del programma scritto e se non incontra errori ne effettua la traduzione dal linguaggio del sorgente al linguaggio proprio del calcolatore (il linguaggio macchina). Ciò che si tiene viene chiamato **programma in versione oggetto**, ma in questo contesto il termine oggetto non assume lo stesso significato a cui abbiamo fatto riferimento parlando di oggetti e classi. Se il sorgente contiene uno o più errori sintattici non viene prodotto il file con la versione oggetto, ma viene fornito l'elenco degli errori riscontrati.

Il programma in versione oggetto è scritto in linguaggio macchina, ma non è ancora immediatamente eseguibile perché mancano alcune ulteriori informazioni e parti comuni che dovranno essere aggiunte. Per questo motivo la fase successiva è di far elaborare il programma in versione oggetto da un altro programma chiamato **linker** (termine inglese che significa 'collegatore') che appunto ha la funzione di collegare i riferimenti mancanti nel programma oggetto al fine di ottenere la versione definitiva chiamata **programma eseguibile**. A volte lo si trova indicata anche come versione binaria (binary).

In questa versione il programma può essere eseguito dal computer producendo gli effetti previsti dalle istruzioni che lo compongono.

Il programma eseguibile ottenuto è applicabile solamente ad un unico tipo di computer e tipo di sistema operativo. Quindi un programma compilato per un pc con sistema operativo windows potrà essere utilizzato dai compiter con queste caratteristiche e quindi non da

uno che abbia ad esempio il sistema operativo linux. Per questo ultimo occorre ri-effettuare i passaggi partendo dal sorgente e utilizzando compilatore e linker specifici per linux.

Appartengono a questa categoria ad esempio i linguaggi C++, Fortran, Cobol e molti altri

Linguaggi interpretati

Per linguaggi di questo tipo non esiste una fase di traduzione del sorgente. Si fa uso di un programma chiamato **interprete** a cui viene affidata l'elaborazione del sorgente. L'interprete esamina la prima istruzione, ne verifica la correttezza sintattica e in caso positivo esegue direttamente l'istruzione stessa, poi passa alla successiva e così via.

Se una istruzione contiene un errore l'interprete interrompe l'esecuzione e segnala a che punto si ferma e il motivo. Quindi un programma contenente uno o più errori potrà essere parzialmente eseguito.

Su computer e/o sistemi opertivi diversi lo stesso sorgente poà essere interpretato a patto di disporre del programma interprete adeguato all'ambiente specifico.

Fanno parte di questa categoria ad esempio i linguaggi Basic, PHP, Perl, Pyton e molti altri

Linguaggi a codice intermedio

Per questa categoria di linguaggi il programma sorgente deve subire una prima fase di verifica sintattica e traduzione producendo una versione detta **codice intermedio**. La differenza rispetto alla compilazione è che ciò che si è ottenuto non è ancora espresso secondo il linguaggio macchina, per essendo una forma non più comprensibile a noi. Il codice intermedio ottenuto deve essere affidato all'elaborazione di una '**macchina virtuale**' che è in grado di interpretare ed eseguire il suo contenuto. La macchina virtuale è in sostanza un programma specifico per tipo di computer e sistema operativo.

In sostanza ci troviamo in una situazione 'mista' delle due precedenti. La prima fase di traduzione è simile a quella del compilatore, ma produce un codice intermedio che è indipendente dal particolare tipo di computer/sistema operativo. Quindi un programma in versione codice intermedio è in sostanza eseguibile su piattaforme diverse. D'altra parte la macchina virtuale è in qualche modo simile ad un interprete, con la differenza che ha maggiore efficienza perché non deve occuparsi di verifica sintattica (già fatta nella traduzione verso il codice intermedio) e la traduzione finale stessa è più semplice visto che si parte da un codice molto più vicino alla macchina del linguaggio di programmazione.

Fanno parte di questa categoria ad esempio i linguaggi Java, C#, Pascal e altri

Il mio primo programma java

Java

Java è il linguaggio di programmazione che utilizzeremo durante questo corso. Come già detto è un linguaggio di tipo a codice intermedio ed è modellato per la programmazione orientata agli oggetti (OOP Object Oriented Programming).

Esistono più di un modo operativo per scrivere programmi sorgenti Java, tradurli ed eseguirli. Inizialmente utilizziamo un ambiente che predispone in modo unitario tutto quanto server per le diverse fasi e viene pertanto chiamato ambiente di sviluppo (in inglese IDE: Integrated Development Environment).

Questo ambiente si chiama NetBeans ed è realizzato dalla società informatica SUN che è anche la società che ha inventato Java. Il programma è scaricabile gratutamente all'indirizzo web http://www.netbeans.org/ .

Scriviamo il primo programma

Ogni scritto dal più formale al più divulgativo che presenti come realizzare il primo programma in una qualunque ambiente propone di realizzare un saluto "Ciao mondo!" e così sarà anche nel nostro caso...

Esplicito in un modo un po' più formale l'obiettivo: intendiamo progettare e realizzare un programma che sia in grado di visualizzare la frase "Ciao mondo!". I passaggi sono :

- scrivere secondo le regole di java il programma, correggendo eventuali errori fino ad ottenere un programma che può essere eseguito
- richiedere l'esecuzione del programma ottenendo l'effetto desiderato

Per arrivarci dobbiamo conoscere alcune caratteristiche di Java.

La dichiarazione di classe

Un primo punto è la regola su come descrivere una classe:

```
class <nomeclasse> {
  <descrizione della classe>
}
```

in cui:

- class è un **parola chiave** (**keyword**) che va sempre scritta esattamente così, serve per indicare che ciò che segue è la dichiarazione di una classe, cioè la descrizione della classe che si sta realizzando
- <nomeclasse> rappresenta genericamente una parola che scegliamo noi arbitrariamente per indicare (identificare) la classe che stiamo dichiarando. Viene chiamato identificatore della classe

Quindi potrà essere:

```
class Ciao {
  <descrizione della classe>
}
```

Va precisato che il linguaggio java prevede la netta distinzione tra caratteri minuscoli e maiuscoli.

Quindi se si scrive Class, o CLASS o altre combinazioni minuscolo/maiuscolo la parola non viene riconosciuta come parola chiave.

Per quanto riguarda invece l'identificatore di classe seguiremo la convenzione che viene tipicamente adottata: l'identificatore di classe ha iniziale maiuscola e resto minuscolo se costituito da una unica parola. Può essere una parola composta da più termini ottenuta scrivendo di seguito, tutto attaccato, le parole componenti, ciascuna con l'iniziale maiuscola. Esempio potremmo chiamare la nostra classe CiaoMondo.

Dopo la parola chiave 'class' e l'identificatore 'Ciao' segue una parentesi graffa aperta '{' a cui corrisponde più avanti la graffa chiusa '}'. Anche questi sono elementi obbligatori che fanno parte della corretta sintassi di scrittura di una classe.

Tra le due graffe la *<descrizione della classe>* rappresenta genericamente quanto effettivamente descrive ciò che la classe contiene (attributi) e/o deve poter fare (metodi).

Metodi di una classe

I metodi di una classe descrivono ciò che desideriamo che la classe sia in grado di fare, rappresentano dunque le capacità di azione della classe.

In generale siamo liberi di progettare quanti metodi riteniamo siano utili ai nostri fini ma nel nostro caso stiamo progettando una classe che ha un compito specifico: rispondere al comando di avvio programma.

Quindi posticipo la descrizione di come comportarsi in modo generico sui metodi e partiamo dal particolare metodo di questa classe.

Il metodo main

Riformulo la proprietà che ci serve.

Ci serve una classe che è in grado di rispondere allo stimolo di avvio del programma (ad esempio il doppio click di un'icona, o un comando scritto in una finestra a comandi).

Per fare in modo che una classe abbia questa caratteristica deve avere un metodo che sia così scritto:

```
public static void main (String[] args) {
  <contenuto del metodo>
}
```

La parte *<contenuto del metodo>* rappresenta genericamente ciò che lo specifico nostro programma dovrà realizzare, le altre parti invece sono 'fisse' cioè vanno scritte esattamente così. Ci chiariremo strada facendo l'esatto significato di ognuno dei termini utilizzati. Per ora vorrei soffermarmi sul perché di questo vincolo su questo metodo.

Quando facciamo eseguire un programma (ad esempio con doppio click su un'icona) c'è una sorta di "passaggio di consegne" in cui il computer prende in carico il programma e incomincia ad eseguirlo. Ma da che punto? Ci vuole un punto "inizia da qui" che sia immediatamente riconoscibile. Bene, nei programmi java "l'inizia da qui" è costituito dal metodo con le regole descritte in precedenza. Per essere completi va anche aggiunto che tale metodo deve essere all'interno di una classe che ha come identificatore lo stesso

nome del programma.

Riassumendo quanto visto fin qui, la classe è così definita:

```
class Ciao {
  public static void main (String[] args) {
    <contenuto del metodo>
  }
}
```

dobbiamo ancora definire cosa mettere nel contenuto del metodo ma intanto salviamo il file con il nome Ciao.java in modo che il programma che otterremo abbia lo stesso nome della classe.

Come descrivere il "contenuto del metodo". In sostanza vogliamo che appaia la scritta 'Ciao mondo!', quindi ci serve un 'qualcosa che 'sa' come scrivere una frase.

Per questo abbiamo a disposizione in java alcune classi ed oggetti "predefiniti" che sono cioè pronti all'uso. Si tratta di ricordarsi che esistono, cosa fanno e come vanno utilizzati... :-)

Nel nostro caso ci viene in aiuto "Sistem.out.println" che è un metodo che abbiamo a disposizione per poter far visualizzare una frase in una finestra di comandi. La frase va specificata come parametro passato al metodo in questo modo: Sistem.out.println("Ciao mondo!"), e dunque ora abbiamo un aspetto completo per la nostra classe:

```
class Ciao {
  public static void main (String[] args) {
    System.out.println("Ciao mondo!");
  }
}
```

La riga che contiene la 'chiamata' del metodo println termina con un ";". Questa regola è generale, ogni istruzione java termina con ";".

Preciso che siccome i passaggi sono già stati tanti ne ho qui saltato uno piuttosto importante che vedremo nel prossimo paragrafo e di cui faremo uso comunque da lì in poi.

Progettare una classe

Avevo anticipato che per raggiungere rapidamente un punto compiuto abbiamo saltato un passaggio.

Il punto è questo: una buona tecnica di sviluppo delle classi consiste nel definire una classe per ogni obiettivo o sotto-obiettivo in modo che ciascuna abbia un proprio ambito di lavoro il più possibile circoscritto, verificabile e componibile con gli altri.

Nel nostro caso separiamo il metodo main che ha la funzione di innesco dalla classe che sviluppa la nostra logica.

Stiamo parlando di azioni estremamente semplici, ma imparare a separare i sotto obiettivi da subito aiuta a farlo anche quando il gioco si fa più duro.

Il passaggio da compiere nel risolvere un problema è “quale oggetto mi aiuta a risolvere il problema?”.

Quindi mi serve una classe con un metodo in grado di compiere lo scopo, e da qui istanziare un oggetto con cui invocare il metodo.

Iniziamo a definire la nuova classe seguendo le regole note:

```
class Saluta {
  <descrizione della classe>
}
```

La classe Saluta deve prevedere un metodo che ha il compito di produrre la visualizzazione di saluto.

Per procedere occorre conoscere le regole di definizione di un metodo di una classe.

Definizione di un metodo

In generale la regola di definizione è:

```
  <modificatori> <identificatoreDiMetodo>(<eventuali parametri>) {
    <contenuto del metodo>
  }
```

L'identificatore del metodo è il nome da dare al metodo per distinguerlo dagli altri. Sulla scrittura dell'identificatore di metodo valgono regole simili a quelle di identificatore di classe salvo che la convenzione tipicamente in uso è che iniziano sempre con lettera minuscola. Può essere una parola composta da più termini ottenuta scrivendo di seguito, tutto attaccato, le parole componenti, ciascuna con l'iniziale maiuscola.

Nel nostro esempio avremo:

```
  <modificatori> visualizza(<eventuali parametri>) {
    <contenuto del metodo>
  }
```

Dopo l'identificatore, tra le parentesi, i “parametri” sono dedicati al eventuale scambio di informazioni nel momento in cui si fa uso del metodo. Per ore non ne facciamo uso e quindi lasciamo vuoto.

I “modificatori” che troviamo al primo posto servono per descrivere alcune caratteristiche del metodo. Per ora ne utilizziamo 2:

- **public** che indica il il metodo può essere liberamente utilizzato. In futuro vedremo altre opzioni alternative a questa.
- **void** che indica che il metodo ha un comportamento di tipo 'azione'. Affronteremo con più precisione nel seguito.

Ecco dunque come descrivere il metodo:

```
  public void visualizza() {
    <contenuto del metodo>
  }
```

Il contenuto del metodo è esattamente lo stesso del metodo main della classe Ciao dell'esempio precedente, per cui la classe Saluta è così completata:

```
class Saluta {
  public void visualizza() {
    System.out.println("Ciao mondo!");
  }
}
```

Ritornano allo sviluppo della nuova classe Ciao vediamo come poter istanziare un oggetto.

Istanziare oggetti

Per istanziare un oggetto questa è la sintassi standard:

```
new <IdentificatoreDiClasse>(<eventuali parametri>);
```

dove

- **new** è la parola chiave che identifica l'operatore che è in grado di istanziare l'oggetto di una classe
- come in precedenza i parametri sono opzionali e al momento non li trattiamo

Nel nostro caso diventa:

```
new Saluta();
```

l'effetto è che viene creato un oggetto della classe Saluta.

Questa operazione da sé non è particolarmente utile, perché non indica cosa fare con questo oggetto appena creato.

La cosa più semplice che possiamo fare è di ottenere la memorizzazione di questo oggetto, cioè far in modo che l'oggetto rimanga a disposizione per il tempo che ci occorre durante l'esecuzione del nostro programma.

Variabili e attributi

La memorizzazione di oggetti e dati è affidata a specifici componenti che possono essere definiti all'interno di un metodo, in tal caso si chiamano variabili, o a livello di classe e in tal caso si chiamano attributi.

Una prima caratteristica di variabili e attributi è di essere tipizzata, cioé ogni variabile si specializza per poter ositare oggetti di una determinata classe su cui sono definiti.

La regola generale della dichiarazione di varibile (o attributo) è :

```
<IdentificatoreDiClasse> <identificatoreDiVariabile>;
```

Per gli identificatori di variabili valgono le stesse regole di scrittura che abbiamo visto per gli identificatore di metodo: iniziano sempre con lettera minuscola. L'identificatore può essere una parola composta da più termini ottenuta scrivendo di seguito, tutto attaccato, le parole componenti, ciascuna con l'iniziale maiuscola.

Con una dichiarazione di questo tipo

Nel nostro caso desideriamo una variabile che sia destinata ad un oggetto di classe saluta, per cui la dichiarazione sarà:

```
Saluta ciao;
```

L'effetto di questa dichiarazione è che 'ciao' è una variabile destinata ad ospitare un oggetto di classe Saluta, ma di fatto non contiene nulla.

E' solo “collegando” l'istanziazione alla variabile che effettivamente si potrà disporre un oggetto con cui svolgere azioni.

Il “collegamento” ci viene fornito dall'operatore di assegnamento.

Assegnamento

L'assegnamento è l'operazione che consente di 'riempire' di contenuto una variabile.

La sua forma sintattica è:

```
<identificatoreDiVariabile> = <espressione>
```

dove '**=**' non sta ad indicare che quello che è scritto a sinistra di '=' è identico a ciò che è scritto a destra, ma che ciò che si ottiene dalla 'espressione' scritta a destra deve essere memorizzato nella variabile indicata a sinistra.

Nel nostro esempio sarà:

```
ciao = new Saluta();
```

E' consentito riassumere in unica riga i due passaggi (dichiarazione di variabile e assegnazione di istanza di oggetto in questo modo:

```
Saluta ciao = new Saluta();
```

Esecuzione di un metodo

Ora che abbiamo una variabile con oggetto di tipo classe (o più sinteticamente diciamo un oggetto di tipo classe) per fare eseguire il metodo abbiamo a disposizione l'operatore '.' (punto).

```
<oggetto>.<metodo>();
```

l'effetto è che viene eseguito il metodo richiesto della classe a cui appartiene l'oggetto.

Nel nostro caso sarà:

```
ciao.visualizza();
```

Seconda versione del primo programma

Ecco dunque sviluppato per intero il primo programma riscritto in questa seconda versione.

```
class Saluta {
  public void visualizza() {
    System.out.println("Ciao mondo!");
  }
}

public class Ciao {
  public static void main(String[] args) {
    Saluta ciao = new Saluta();
    ciao.visualizza();
  }
}
```

Indentazione

Osservando il codice appena scritto si può notare che le riche non sono tutte allineate a sinistra ma sono presenti degli 'sfasamenti' verso destra, che prendono il nome di **indentazione**.

Questa tecnica di presentazione del codice sorgente si basa sul presentare incolonnate allo stesso livello elementi “pari” tra di loro, di spostare verso destra un elemento contenuto in un altro.

Infatti troviamo all'estrema sinistra le definizioni di classe, più a destra le definizioni di metodo contenute nelle rispettive classi e più a destra ancora le istruzioni interne ad ogni metodo.

Riguardo alle parentesi graffe che in generale “racchiudono” qualcosa una convenzione 'tipica” (anche se non unica) è che la parentesi graffa aperta è scritta all'estremo destro della riga “contenitore” e analogamente la parentesi graffa chiusa viene incolonnata al livello pari della riga “contenitore”.

Esercizi ed approfondimenti

Ripetizione di consolidamento

Scrivere un programma simile al precedente (con le due classi) in cui la frase visualizzata è 'Ciao lettore!”

Chiamata successiva di metodo

Modificare il programma CiaoMondo scrivendo in tre righe di seguito l'invocazione del metodo ciao.visualizza(). Che accade?

Realizzazione di classe

Scrivere una classe 'Autore' con un metodo che permette di visualizzare “Questo programma è stato scritto da Mario Rossi” (dove sostituirai con il tuo nome e cognome) e poi una classe con metodo main per verificare che funzioni correttamente

Uso di due classi

Modificare il primo programma Ciao mondo con le due classi aggiungendo come terza classe Autore, il metodo main dovrà disporre di due oggetti uno di classe Salute a l'altro di classe Autore, con questi visualizzare prima il saluto e poi la nota di autore.

Approccio a un problema

Produrre il programma 'Biglietto da visita' Che visualizza il testo:

Mario Rossi

Classe 3X5

ITIS Belluzzi Bologna

Classi e oggetti con più metodi

Eseguire un metodo più volte

Possiamo richiedere ad un oggetto di eseguire più volte un metodo, ottenendo quindi più volte l'effetto prodottoo da quel metodo.

Ecco come vedere tre volte il saluto:

```
package saluto;

public class Saluto {
  public void saluta(){
      System.out.println("Ciao, mondo!");
  }
}
```

```
package saluto;

public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        //preparo una variabile per oggetto di classe Saluto
        Saluto ciao;
        //istanzio un oggetto di classe Saluto
        ciao = new Saluto();
        //l'oggetto effettua il saluto per tre volte
        ciao.saluta();
        ciao.saluta();
        ciao.saluta();
    }

}
```

Si otterrrà questo output:

```
Ciao, mondo!
Ciao, mondo!
Ciao, mondo!
```

Classe con più metodi

Possiamo arricchire la classe con più metodi in modo da poter ottenere comportamenti diversi, ad esempio:

```
package saluto;
```

```
public class Saluto {
  public void saluta(){
      System.out.println("Ciao, mondo!");
  }
  public void salutaClasse(){
      System.out.println("Ciao, 3Ai!");
  }
  public void salutaProf(){
      System.out.println("Ciao, professore!");
  }
  public void salutaMe(){
      System.out.println("Ciao, Gianni!");
  }}
```

La classe Saluto offre ora anche altri tre metodi che realizzano messaggi di saluto di tipo diverso.

Il metodo main con un oggetto di questa classe può chiederne l'esecuzione.

```
package saluto;

public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        //preparo una variabile per oggetto di classe Saluto
        Saluto ciao;
        //istanzio un oggetto di classe Saluto
        ciao = new Saluto();
        //l'oggetto effettua vari saluti
        ciao.saluta();
        ciao.saluta();
        ciao.salutaClasse();
        ciao.salutaProf();
        ciao.salutaMe();
    }

}
```

Si otterrrà questo output:

```
Ciao, mondo!
Ciao, mondo!
Ciao, 3Ai!
Ciao, professore!
Ciao, Gianni!
```

Si noti che si è liberi di eseguire i metodi nell'ordine che si preferisce, indipendentemente

dall'ordine con cui sono definiti nella classe, ad esempio:

```
package saluto;

public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        //preparo una variabile per oggetto di classe Saluto
        Saluto ciao;
        //istanzio un oggetto di classe Saluto
        ciao = new Saluto();
        //l'oggetto effettua vari saluti
        ciao.salutaClasse();
        ciao.saluta();
        ciao.salutaMe();
        ciao.saluta();
        ciao.salutaProf();
    }

}
```

produce questo output:

```
Ciao, 3Ai!
Ciao, mondo!
Ciao, Gianni!
Ciao, mondo!
Ciao, professore!
```

Metodi con parametri

Stesso metodo, effetti diversi

Abbiamo visto che possiamo ottenere effetti diversi con metodi diversi. In questo modo ogni volta che si desidera un nuovo effetto si modifica la classe aggiungendo un metodo.

Osserviamo che le azioni dei metodi realizzati nei precedenti esempi sono tutte molto simili: producono la visualizzazione di un messaggio, la differenza sta nel contenuto di ogni specifico messaggio.

Per casi come questi è possibile definire metodi per i quali si stabilisce un comportamento che può essere applicato a dati diversi: metodi con parametri.

Formule e parametri

Prima di passare ai metodi con parametri facciamo riferimento ad una esperienza ormai consolidata dallo studio della matematica in cui normalmente esprimiamo proprietà in modo parametrico. Ad esempio se facciamo un acquisto di 2 kg di mele al prezzo di 0,8 euro al chilo sappiamo che il costo è 2x0,8 = 1,6 euro.Se invece l'acquisto riguarda 2,5 kg di pere al prezzo di 1,2 euro al kilo il costo è 2,5x1,2 = 3 euro.

Potremmo proseguire trattando ogni singolo caso, ma sappiampo che possiamo esprimere la formula in modo parametrico: il costo di merce acquistata è pari a **peso** x **prezzo**. Di volta in volta si sostituisce a peso e prezzo il relativo valore e si ottiene il risultato che interessa.

I metodi, i problemi e gli algoritmi

Problema e algoritmo

Il metodo di una classe ha il compito di raggiungere un certo obiettivo che potremmo anche vedere in ottica di problema...

Modello di problema Dati iniziali – Soluzione – Risultato – Verifica...

La soluzione come algoritmo e sue caratteristiche

I dati semplici e loro rappresentazione

variabili e costanti

String

integer

operazioni, operatori espressioni

real

Esempi ed esercizi

Biglietto da visita

Il problema del capitolo precedente in cui i dati propri del problema sono in attributi della classe biglietto

Rispetto alla prima soluzione si spostano in attributi della classe i dati specifici

```
/**
 * Classe per la visualizzazione di saluto
 *
 * @author gianni
 */
class Biglietto {
  String nomeCognome="MarioRossi";
  String classe="3X5";
  String scuola="ITIS Belluzzi";
  /**
   * Visualizza Nome e Cognome
   */
  public void visualizzaNomeCognome() {
    System.out.println(nomeCognome);
  }
  /**
   * Visualizza Classe
   */
  public void visualizzaClasse() {
    System.out.println("Classe "+classe);
  }
```

```
  /**
   * Visualizza Nome e Cognome
   */
  public void visualizzaScuola() {
    System.out.println(scuola);
  }
}

/**
 *
 * @author gianni
 */
public class ScriviBiglietto {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Biglietto presentati = new Biglietto();
        presentati.visualizzaNomeCognome();
        presentati.visualizzaClasse();
        presentati.visualizzaScuola();
    }

}
```

Classi ed oggetti come modello per rappresentare il problema

Abbiamo fin qui trattato programmi di tipo 'comunicativo', in cui cioè il programma è in grado di presentare, al comando, informazioni già note da no fin dall'inizio.

Passiamo a trattare ora programmi che aggiungono anche funzioni di tipo 'elaborativo' in cui cioè il programma parte da alcune informazioni iniziali, le rielabora, produce una nuova informazione e la comunica.

Metodi per eseguire algoritmi

metodi cablati su 'un caso'

I dati interi

```
/*
 * La mamma di Pierino compra 4 scatole di cioccolatini,
 * ogni sctola contiene 8 cioccolatini.
 * Quanti cioccolatini ha preso in tutto?
 */

/**
 * Classe per il calcolo dei pezzi totali
```

```
 * @author gianni
 */
class CalcolatricePezzi {
  int numeroScatole;
  int pezziPerScatola;
  int pezziTotali;
  public void imposta() {
    pezziPerScatola=8;
    numeroScatole=4;
  }
  public void calcola(){
    pezziTotali=pezziPerScatola*numeroScatole;
  }
  public void visualizza(){
    System.out.print("Il numero totale di pezzi è ");
    System.out.println(pezziTotali);
  }
}

/**
 *
 * @author gianni
 */
public class Cioccolatini {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        CalcolatricePezzi spesa = new CalcolatricePezzi();
        spesa.imposta();
        spesa.calcola();
        spesa.visualizza();
    }

}
```

I dati 'con la virgola'

```
/*
 * Pierino dal fruttivendolo compra 2,3 Kg di mele che costano
 * 1,2 euro/Kg. Quanto spende in tutto?
 */

/**
 * Classe per il calcolo della spesa
 * @author gianni
 */
```

```
class Calcolatrice {
  double quantita;
  double prezzo;
  double spesa;
  public void imposta() {
    prezzo=1.2;
    quantita=2.3;
  }
  public void calcola(){
    spesa=quantita*prezzo;
  }
  public void visualizza(){
    System.out.print("La spesa è ");
    System.out.println(spesa);
  }
}

/**
 *
 * @author gianni
 */
public class Pierino {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Calcolatrice spesa = new Calcolatrice();
        spesa.imposta();
        spesa.calcola();
        spesa.visualizza();
    }

}
```

generalizzazione e necessità di parametrizzare

Metodi con parametri di input

Metodi con valore out

parametri in/out

Problema scomposto in classi

Separazione dei compiti alle classi:

- classe che rappresenta un certo dato e le sue azioni proprie
- classi per i/o
- classe main che implementa le fasi di lavoro

Esempio:

```
package rettangolo;

/**
 *
 * @author gianni
 */
class Visualizzatore {
  /**
   * Visualizza la stringa passata
   *
   * @param s la stringa da visualizzare
   */
  public void scrivi (String s){
    System.out.println(s);
  }
}

/**
 * Rappresentazione di un trettangolo e sue proprietà
 *
 * @author gianni
 */
class Rettangolo {
  /**
   * la base
   */
  double base=3.2;
  /**
   * l'altezza
   */
  double altezza=5.7;

  /**
   * calcolo del perimetro
   *
   * @return il valore del perimetro
   */
  public double perimetro(){
    return 2*(base+altezza);
  }

  /**
   * calcolo dell'area
   *
   * @return il valore dell'area
   */
  public double area() {
    return base*altezza;
  }
}
```

```
/**
 *
 * @author gianni
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
      Rettangolo r=new Rettangolo();
      Visualizzatore v= new Visualizzatore();
      v.scrivi("Calcoli su un rettangolo");
      v.scrivi("Perimetro = "+r.perimetro());
      v.scrivi("Area = "+r.area());
    }

}
```

Il costruttore

Istanza di oggetto e inizializzazione dei suoi dati

personalizzazione del costruttore

overload del costruttore

Information hiding

Il modificatore private

Soluzione di problemi

Classi di interfaccia

Necessità di parametrizzare l'intero programma

Classi per I/O tastiera e monitor

Esempi di i/o in console e in finestra (?)

Sviluppo sw

Dal problema al progetto

scomposizione

imlementazione

tracciatura dell'esecuzione

funzioni dell'ambiente di sviluppo per la tracciatura

validazione

dati di prova

Nascondere dettagli implementativi

Partiamo dal progetto realizzato AtletiGUI (http://amplio00.belluzzi.scuole.bo.it/mod/resource/view.php?id=76549)

In questo pregetto la classe model SquadraDiAtleti ha il compito di gestire i dati degli atleti di una squadra. Sappiamo che internamente gli atleti sono organizzati in un array, e questo traspare anche al di fuori perché i due metodi

```
public void setAtleta(int i, Atleta a){...}
public Atleta getAtleta(int i){...}
```

richiedono l'esplicitazione dell'indice del dato a cui riferirsi.

Analizziamo il contesto d'uso del metodo getter possiamo osservare che viene utilizzato durante una iterazione su tutti i dati in squadra:

```
//scrivi su file tutti gli atleti della squadra
for (int i=0;i<atleti.getNumeroAtleti();i++){
    scrittore.setPettorale(atleti.getAtleta(i).getPettorale());

scrittore.setCognomeNome(atleti.getAtleta(i).getCognomeNome());
    scrittore.setPeso(atleti.getAtleta(i).getPeso());
}
```

Possiamo raggiungere allo stesso obiettivo senza esplicitare l'uso di un indice se disponiamo di un oggetto che permette di “scorrere” (iterare) lungo tutti i dati garantendoci di poter passare ciclicamente su di tutti senza saltarne alcune né mai ripeterne.

Un oggetto di questo tipo prende il nome di iteratore e permette di raggiungere l'obiettivo prefissato se dispone di due metodi:

```
public boolean hasNext()
```

che ritorna true se sono disponibili altri dati, e

```
public Atleta next()
```

che restituisce il 'prossimo' atleta.

Ecco una possibile implementazione:

```
public class SquadraIterator {
  private SquadraDiAtleti squadra;
  int indice=0;

  public SquadraIterator(SquadraDiAtleti squadra) {
    this.squadra = squadra;
  }



  public boolean hasNext(){
```

```
    return indice<squadra.getNumeroAtleti();
  }
  public Atleta next(){
    if (hasNext()){
      Atleta a=squadra.getAtleta(indice);
      indice++;
      return a;
    }
    else return null;
  }
}
```

e quindi il segmento di codice del precedente esempio diventerebbe:

```
//scrivi su file tutti gli atleti della squadra
SquadraIterator iteratore= new SquadraIterator(atleti);
while (iteratore.hasNext()){
  Atleta a=iteratore.next();
  scrittore.setPettorale(a.getPettorale());
  scrittore.setCognomeNome(a.getCognomeNome());
  scrittore.setPeso(a.getPeso());
}
```

come si vede non è stato più necessario esplicitare l'uso dell'indice.

Rimane il fatto che fino a che la classe SquadraDiAtleti rende pubblico l'uso dell'indice non avremo reso totalmente invisibili gli elementi implementativi, ma d'altra parte la classe SquadraIterator deve poter accedere all'indice per poter svolgere il suo compito. Se potessimo rendere la classe SquadraIterator un elemento interno alla classe SquadraDiAtleti allora non sarebbe più necessario rendere visibile l'uso dell'indice.

Questo è possibile definendo SquadraIterator come **inner class** della classe SquadraDiAtleti, cioè scrivendo fisicamente la sua definizione all'interno della classe squadra di Atleti.

```
public class SquadraDiAtleti {
  private Atleta[] squadra;

//...

  /**
   * getter dell'atleta di indice i
   * @param i indice, non viene testato il fuori range
   * @return atleta
   */
  private Atleta getAtleta(int i){
    return squadra[i];
  }
  // ...
```

```
  public class SquadraIterator {
    private SquadraDiAtleti squadra;
    int indice=0;

    public SquadraIterator(SquadraDiAtleti squadra) {
      this.squadra = squadra;
    }



    public boolean hasNext(){
      return indice<squadra.getNumeroAtleti();
    }
    public Atleta next(){
      if (hasNext()){
        Atleta a=squadra.getAtleta(indice);
        indice++;
        return a;
      }
      else return null;
    }
  }

}
```

Si noti che il metodo getAtleta(indice) è divento privato.

La sintassi d'uso di questa classe diventa:

```
//scrivi su file tutti gli atleti della squadra
SquadraDiAtleti.SquadraIterator iteratore= atleti.new
SquadraIterator(atleti);
while (iteratore.hasNext()){
  Atleta a=iteratore.next();
  scrittore.setPettorale(a.getPettorale());
  scrittore.setCognomeNome(a.getCognomeNome());
  scrittore.setPeso(a.getPeso());
}
```

Si osservi ora che in realtà non c'è più bisogno che l'iteratore abbia in sé un riferimento all'oggetto SquadraDiAtleti; dal momento che ne fa parte internametne può accedere direttamente al suo array di dati:

```
public class SquadraDiAtleti {
  private Atleta[] squadra;

//...
```

```
  public class SquadraIterator {
    int indice=0;

    public boolean hasNext(){
//       return indice<squadra.getNumeroAtleti();
      return indice<squadra.length;
    }
    public Atleta next(){
      if (hasNext()){
        Atleta a=squadra[indice];
        indice++;
        return a;
      }
      else return null;
    }
  }
}
```

si noti che ora il metodo getAtleta può essere rimosso.

Per rimuovere consideriamo che per la funzione di modifica del dato abbiamo già visto che non serve, rimane l'uso che ne abbiamo fatto al momento di caricare i dati letti da file:

```
  public static SquadraDiAtleti popolaAtletiViaFile(File f,
AtletiConsoleView console) throws Exception {
    SquadraDiAtleti atleti;

    // viene aperto il file e determinato numAtleti

    atleti=new SquadraDiAtleti(numAtleti);
    //controllo la lettura con ciclo sul numero di atleti
    for (int i =0; i< numAtleti ; i++ ){
      // ...
      //leggo da file pettorale, cn, peso
      // ...
      atleti.setAtleta(i, new Atleta(pettorale,cn,peso));
    }
    //lettura completa
    return atleti;
  }
```

Qui di fatto si stanno di volta in volta aggiungendo nuovi elementi, quindi esplicitiamo questo con un metodo aggiungi, questo comporta la necessità di avere un contatore di dati effettivamente presenti nell'array.

```
/**
 * Gestione dei dati di una squadra di atleti
 * @author Gianni
 */
public class SquadraDiAtleti {
  private Atleta[] squadra;
  private int numeroAtleti;
```

```
  /**
   * Costruttore, istanzia la squadra su un numero fornito di
atleti
   * @param numAtleti il numero di altleti su cui dimensionare
   */
  public SquadraDiAtleti(int numAtleti) {
    squadra = new Atleta[numAtleti];
    numeroAtleti=0;
  }

  /**
   * Costruttore, referenzia un array di atleti passato
   * @param squadra array da referenziare
   */
  public SquadraDiAtleti(Atleta[] squadra) {
    this.squadra=squadra;
    numeroAtleti=squadra.length;
  }

  /**
   * Conversione a string dei dati di un atleta
   * @return dati dell'atleta
   */
  @Override
  public String toString() {
    String out="";

    out += "SquadraDiAtleti{" + "\nlunghezza=" + squadra.length;
    for (int i = 0; i<squadra.length ; i++){
      out += "\natleta["+i+"]="+squadra[i]+";";
    }
    out+=  "\n}\n";
    return out;
  }

  /**
   * Aggiunge un atleta
   * @param a il dato da agggiungere
   * @throws Exception spazio esaurito
   */
  public void aggiungi(Atleta a)throws Exception {
    if (numeroAtleti<squadra.length){
      squadra[numeroAtleti]=a;
      numeroAtleti++;
    }
    else {
      throw new Exception("Spazio esaurito in squadra");
    }
  }
```

```
  /**
   * Ricerca nei dati l'atleta corrispondente al numero di
pettorale passato
   * @param pettorale da ricercare
   * @return atleta trovato
   * @throws Exception non esiste il pettorale ricercato
   */
  public Atleta ricercaPettorale(int pettorale) throws Exception{
    for (int i = 0; i<squadra.length ; i++){
      if (squadra[i].getPettorale()==pettorale){
        return squadra[i];
      }
    }
    throw new Exception();
  }

  /**
   * Il numero dei dati effettivamente presenti
   * @return
   */
  public int getNumeroAtleti(){
      return numeroAtleti;
  }

  /**
   * Classe iteratore sulla squadra
   */
  public class SquadraIterator {
    int indice=0;

    /**
     * ritorna tru ese il prossimo next() avrà successo
     * @return
     */
    public boolean hasNext(){
      return indice<numeroAtleti;
    }

    /**
     * ritorna il prossimo dato
     * @return
     */
    public Atleta next(){
      if (hasNext()){
        Atleta a=squadra[indice];
        indice++;
        return a;
      }
      else return null;
    }
  }
```

```
}
```

Si noti l'eccezione prevista nel metodo aggiungi quando l'array si satura.

Il metodo di caricamento dati da file diventa:

```
  public static SquadraDiAtleti popolaAtletiViaFile(File f,
AtletiConsoleView console) throws Exception {
    SquadraDiAtleti atleti;

    atleti=new SquadraDiAtleti(numAtleti);
    //leggo i dati da file e memorizzo in atleti
    while (lettoreAtleti.ancoraDati()){
      // ...
      //leggo da file pettorale, cn, peso
      //...
      //dati di atleta completi, istanzio oggetto e lo aggiungo in
squadra
      atleti.aggiungi(new Atleta(pettorale,cn,peso));//l'eventuale
eccezione viene portata al livello superiore
    }
    //lettura completa
    return atleti;
  }
```

Interfacce Iterator e Iterable

Il problema appena affrontato è così frequente che Java lo ha "standardizzato" prevedendo classi "generiche" Iterator e Iterable

Il termine **Interfaccia** indica una definizione di classe di cui si dichiara la firma dei metodi ma non il relativo sviluppo di codice.

Sarà compito di altre classi **implementare** l'interfaccia sviluppandone tutti i metodi.

Nell'implementazione è obbligatorio implementare tutti i metodi previsti dall'interfaccia, mentre è liberamente possibile prevedere altri metodi

```
interface MiaInterfaccia{
  public void faiQualcosa(); //NOTA BENE: niente codice del metodo
}
class MiaImplementazioneChiaccherona implements MiaInterfaccia {
  public void faiQualcosa(){
    System.out.println("Io parlo!");
  }
}
class MiaImplementazioneSilenziosa implements MiaInterfaccia {
  int conta=0;
  public void faiQualcosa(){
    conta++;
  }
  public int quantoHaiFatto(){
    return conta;
  }
}
```

Per essere più precisi le interfacce Iterator e Iterable sono parametrizzate, cioè vanno applicate ad una classe; la scrittura più appropriata è Iterator<E> e Iterable<E> dove al posto di E andrà di volta in volta la classe di riferimento, quasi come se fosse un parametro di un metodo.

Interfaccia Iterator<E>

http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

Per essere più concreti nell'esempio di squadre e atleti la classe di riferimento è Atleta, per cui la nostra classe iteratore diventa:

```
class SquadraIterator implements Iterator<Atleta> {
  //il codice di implementazione
}
```

i metodi da implementare sono tre di cui due (non a caso) sono hasNext() e next() che abbiamo già visto, il terzo remove() è destinato alla rimozione dell'elemento corrente, ed ecco una possibile implementazione:

```
  /**
   * Classe iteratore sulla squadra
   */
  class SquadraIterator implements Iterator<Atleta> {
```

```
    int indice=0;
    boolean enableRemove=false;

    /**
     * ritorna tru ese il prossimo next() avrà successo
     * @return
     */
    public boolean hasNext(){
      enableRemove=indice<numeroAtleti;
      return enableRemove;
    }

    /**
     * ritorna il prossimo dato
     * @return
     */
    public Atleta next(){
      if (hasNext()){
        Atleta a=squadra[indice];
        indice++;
        return a;
      }
      else {
        throw new NoSuchElementException();
      }
    }

    public void remove(){
      if (enableRemove){
        for (int i=indice; i<numeroAtleti;i++){
          squadra[i]=squadra[i+1];
        }
        squadra[numeroAtleti]=null;
        numeroAtleti--;
        enableRemove=false;
      }
      else {
        throw new IllegalStateException();
      }
    }
  }
```

Interfaccia Iterable<E>

http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html

Ci si aspetta dunque che ogni classe che implemtenta Iterable abbia un metodo iterator() che restituisce un iteratore sugli elementi interni alla classe stessa,

quindi nel nostro caso diventa:

```
/**
 * Gestione dei dati di una squadra di atleti
 * @author Gianni
 */
public class SquadraDiAtleti implements Iterable<Atleta>{

  // . . .

  /**
   * ritorna un iteratore
   * @return
   */
  public Iterator<Atleta> iterator(){
    return new SquadraIterator();
  }

  /**
   * Classe iteratore sulla squadra
   */
  private class SquadraIterator implements Iterator<Atleta> {

    // . . .

  }
}
```

Si noti che la classe SquadraIterator è diventata privata e che il metodo iterator() dichiara di restituire Iterator<Atleta>.

Dal momento che l'interfaccia di Iterator è nota non ho alcun bisogno di conoscere i dettagli di implementazione per utilizzarla, anzi sabendo che un oggetto è di classe Iterator so esattamente che ho disponibili i tre metodi hasNext(), next() e remove(), mentre per un oggetto di classe SquadraIterator ho bisogno di esaminare il suo elenco di metodi per poterne conoscere le properietà.

Proprietà di un oggetto di classe Iterable

La potenza delle interface sta nel fatto che consentono di generalizzare un problema, di permettere di scrivere codice prima che ancora sia stata pensata una implementazione, di scrivere codice applicabile a tutte le possibili implementazioni.

Vediamo un esempio concreto. Per definizione stessa Iterable si sa che un oggetto di una classe che lo implementaavrà un metodo che restituisce un Iterator, e ancora si è certi che quell'Iterator ha a disposizione il metodo hasNext() e next(), per cui la java virtual machine rende disponibile la possibilità di utilizzare un for applicato ad un oggetto di classe Iterable:

```
for (ElementClass element:IterableObject) {
  // utilizzo element
}
```

Nel nostro esempio vediamo il segmento di codice di scrittura su file di tutti i dati di squadra:

```
        Iterator<Atleta> iteratore= atleti.iterator();
```

```
        while (iteratore.hasNext()){
          Atleta a=iteratore.next();
          scrittore.setPettorale(a.getPettorale());
          scrittore.setCognomeNome(a.getCognomeNome());
          scrittore.setPeso(a.getPeso());
        }
```

la classe degli elementi è Atleta, l'oggetto di classe Iterable è atleti, all'interno del ciclo ci serve un oggetto atleta, quindi possiamo scrivere:

```
        for (Atleta a:atleti){
          scrittore.setPettorale(a.getPettorale());
          scrittore.setCognomeNome(a.getCognomeNome());
          scrittore.setPeso(a.getPeso());
        }
```

questo codice è più semplice da scrivere e rende “invisibile” l'iteratore lasciando evidente quello che è più interessante per l'algoritmo, cioè che partiamo dall'oggetto atleti e lavoriamo su ogni (**foreach**) atleta. Inoltre l'interprete della JVM traduce in modo più efficiente il costrutto così ottenuto.

Classi Generics

Nel definire una classe (o un'interfaccia) si può far riferimento a classi generiche , cioè che hanno la funzione di “parametro” a cui si fa riferimento in modo generico e che verranno specificate solo al momento della compilazione.

```
/**
 * Permette di incapsulare un dato che sia di una classe generica
 * @author giovanni.ragno
 */
public class Contenitore<E> {
    E dato;

    /**
     * Costruttore
     * @param dato il dato da memorizzare
     */
    public Contenitore(E dato) {
        this.dato = dato;
    }

    /**
     * Getter
     * @return il dato contennto
     */
    public E getDato() {
        return dato;
    }

    /**
     * Setter
     * @param dato il dato da memorizzare
     */
    public void setDato(E dato) {
        this.dato = dato;
    }

    /**
     * Overridde di toString, utile per le verifiche in sviluppo
     * @return
     */
    @Override
    public String toString() {
        return "Contenitore{" + "dato=" + dato + '}';
    }
}
```

Un semplice test d'uso potrà essere:

```
public class TestGeneric {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Contenitore<String> cs; //memorizza String
        Contenitore<Integer> ci; //memorizza Integer
        Contenitore<Double> cd; //memorizza Double
        cs=new Contenitore<String>("Ciao");//istanza di un
contenitore di String
        System.out.println(cs);//uso del toString
        System.out.println(cs.getDato());//uso del getter
        cs.setDato("Liberi!");//uso del setter
        System.out.println(cs);
        ci=new Contenitore<Integer>(5);//istanza di un contenitore
di Integer
        System.out.println(ci);
        cd= new Contenitore<Double>(3.5*ci.getDato());//istanza di
un contenitore di Double
        System.out.println("cd="+cd);    }
}
```

la suo esecuzione comporta output:

```
Contenitore{dato=Ciao}
Ciao
Contenitore{dato=Liberi!}
Contenitore{dato=5}
cd=Contenitore{dato=17.5}
```

Le classi parametrizzata possono essere più di una, ecco un esempio con due:

```
/**
 * Esempio di classe con due generic
 * @author giovanni.ragno
 */
public class Coppia<E1,E2> {
    E1 primo;
    E2 secondo;

    /**
     * costruttore
     * @param primo
     * @param secondo
     */
    public Coppia(E1 primo, E2 secondo) {
        this.primo = primo;
        this.secondo = secondo;
    }
```

```
    /**
     * getter primo dato
     * @return
     */
    public E1 getPrimo() {
        return primo;
    }

    /**
     * setter primo dato
     * @param primo
     */
    public void setPrimo(E1 primo) {
        this.primo = primo;
    }

    /**
     * getter secondo dato
     * @param primo
     */
    public E2 getSecondo() {
        return secondo;
    }

    /**
     * setter secondo dato
     * @param primo
     */
    public void setSecondo(E2 secondo) {
        this.secondo = secondo;
    }

    /**
     * Overridde di toString, utile per le verifiche in sviluppo
     * @return
     */
    @Override
    public String toString() {
        return "Coppia{" + "primo=" + primo + ", secondo=" +
secondo + '}';
    }

}
```

e un semplice test:

```
    public static void main(String[] args) {
        Coppia<String,String> css; //coppia di String e String
        Coppia<String,Integer> csi; //Coppia di String e Integer
        Coppia<Double,Integer> cdi; //coppia di Double e Integer
```

```
        Coppia<Integer,Contenitore<String>> cics; //coppia di
Integer e Contenitore di String
        Coppia<Integer,Coppia<String,Double>> cicsd; //coppia di
Integer e Coppia di String e Double
        css=new Coppia<String,String>("Ciao","Mondo!");
        System.out.println("css="+css);
        csi=new Coppia<String,Integer>("Test",-3);
        System.out.println("csi="+csi);
        cdi=new Coppia<Double,Integer>(45.2,34);
        System.out.println("cdi="+cdi);
        Contenitore<String> cs=new Contenitore<String>("Ciao");
        cics= new Coppia<Integer,Contenitore<String>>(10,cs);
        System.out.println("cics"+cics);
        cicsd=new Coppia<Integer,Coppia<String,Double>> (12,new
Coppia<String,Double>("Bello, ma inutile!",-32.1e4));
        System.out.println("cicsd="+cicsd);
    }
}
```

la sua esecuzione produce il seguente output:

```
css=Coppia{primo=Ciao, secondo=Mondo!}
csi=Coppia{primo=Test, secondo=-3}
cdi=Coppia{primo=45.2, secondo=34}
cicsCoppia{primo=10, secondo=Contenitore{dato=Ciao}}
cicsd=Coppia{primo=12, secondo=Coppia{primo=Bello, ma inutile!,
secondo=-321000.0}}
```

ADT – Abstract Data Type

Con il termine ADT – Abstract Data Type (Tipo di dato astratto) si intende la definizione di un dato basandosi sulle sue proprietà operative e di manipolazione senza preoccuparsi di quale sia la effettiva implementazione.

In realtà la definizione di una classe di per sé è già una implementazione di un dato incapsulando i dati base e i dettagli implementativi e rendendo pubblico solo le interfacce offerte dai metodi.

La possibilità di parametrizzare classi permette di costruire classi anco più generiche che definiscono proprietà su oggetti e non si semplici dati.

Di un ADT si possono realizzare diverse implementazioni che garantiranno le stesse operatività, ma potranno avere caratteristiche di efficienza diverse sotto diversi aspetti.

Nel seguito vediamo alcuni esempi di ADT e loro implemnentazioni che renderanno più concrete le affermazioni qui fatte.

ADT Sequenza

Ripensiamo all'esercizio già svolto sulla gestione dei dati di una squadra di atleti (vedi http://amplio.belluzzifioravanti.it/mod/assign/view.php?id=74923).

In questo contesto la classe SquadraDiAtleti in sostanza prevede un array di Atleta e un altro dato intero che rappresenta il numero di atleti effettivamente presenti, che può anche variare nel tempo. In realtà molti problemi possono prevedere una situazione simile, per cui può valere la pena generalizzare questo tipo di gestione.

Sequenza

Intendiamo con sequenza una struttura di dati tutti dello stesso tipo, costituito da una successione ordinata dei dati. L'ordinamento è in relazione alla posizione, cioè ad ogni elemento è associato un numero d'ordine (indice) che convenzionalmente facciamo partire da 0.

La sequenza può essere vuota. Sulla sequenza definiamo le seguenti operazioni:

- inserisci un dato d alla posizione i: il dato va in quella posizione facendo scorrere in avanti di un posto quelli che erano presenti da quella posizione in poi, la sequenza si allunda di uno
- appendi un dato d: il dato viene messo dopo quello presente all'ultimo posto, la sequenza si allunga di 1
- elimina il dato alla posizione i: il dato alla posizione i viene “tolto” e il “buco” viene chiuso da quello che segue, la sequenza si accorcia di uno
- get elemento alla posizione i: si ottiene il dato alla posizione i
- test se la sequenza vuota. Si ottiene true se la sequenza è vuota, false se è presente almeno un dato
- get lunghezza: si ottiene il numero dei dati presenti

Definizione della interfaccia

In java possiamo formalizzare la descrizione appena fatta in italiano definendo una interface:

```
/**
 * Una sequenza generica
 * @author Gianni
 */
public interface Sequenza<T> {
  /**
   * il dato viene inserito alla posizione i
   * @param dato da memorizzare
   * @param i posizione
   */
  void inserisci(T dato, int i);
  /**
```

```
   * il dato viene memorizzato dopo l'ultimo
   * @param dato da memorizzare
   */
  void appendi(T dato);
  /**
   * rimuove il dato alla posizione i
   * @param i posizione da cui rimuovere
   */
  void elimina(int i);
  /**
   * ottieni il dato alla posizione i
   * @param i posizione da ottenere il dato
   * @return il dato ottenuto
   */
  T getElemento(int i);
  /**
   * test se la sequenza è vuota
   * @return true se vuota, false in caso contrario
   */
  boolean isEmpty();
  /**
   * la lunghezza della sequenza
   * @return il numero di dati presenti
   */
  int getLunghezza();
}
```

Implementazione di sequenza

È possibile realizzare una classe di implementazione di sequenza facendo uso di un array e pertanto la chiameremo SequenzaAdArray.

ADT stack

Lo **stack** viene detto anche **pila** o **catasta** o ancora **coda LIFO** (Last In First Out) e possiamo figuracela proprio come un un mucchio di oggetti a cui possiamo aggiungere oggetti solo appoggiandoli sulla sommità e viceversa è possibile prendere solo il primo in alto. La conseguenza è che il primo oggetto disponibile è l'ultimo inserito, in altre parole l'ordine di estrazione è l'inverso dell'ordine di inserzione.

Lo stack è una struttura di dati molto utilizzata in ambito informatico, già abbiamo citato che ad ogni chiamata di esecuzione di metodo viene caricato in stack il record di attivazione con tutto il suo ambiente.

Proprietà di stack

Lo stack può essere vuoto, e quindi da uno stack vuoto non è possibile estrarre dati. All'estremo opposto potrà succedere che uno stack abbia saturato lo spazio di memoria disponibile e quindi non sarà possibile inserire ulteriori dati. Inoltre lo stack struttura dati tutti dello stesso tipo. Dopo questa premessa ecco le possibili operazioni di stack.

- Inserzione di un dato d. A differenza di quanto fatto per la sequenza non si indica il punto di inserzione perché come già detto il punto di inserzione deve essaere la cima dello stack. Questa operazione viene chiamata **push**.
- Elimina il dato in cima allo stack. Questa operazione chiamata **pop**.
- Get del valore in cima allo stack. Questa operazione viene chiamata **top**. In alcune implementazioni si può trovare una unica operazione che congiuntamente realizza top e pop, ma noi le terremo distinte.
- Test se lo stack è vuoto.

Definizione della interfaccia

Ecco la formalizzazione in java dell'elenco delle operazioni descritte:

```
/**
 * Proprietà di stack
 * @author giovanni.ragno
 */
public interface Stack<T> {
  /**
   * inserisce un dato in cima allo stack
   * @param dato da inserire
   */
  void push(T dato);
  /**
   * elimina il dato in cima allo stack
   */
  void pop();
  /**
   * restituisce il dato in cima allo stack
   * @return il dato
   */
```

```
  T top();
  /**
   * test se lo stack è vuoto
   * @return true se vuoto, false in caso contrario
   */
  boolean isEmpty();
}
```

Implementazione di stack

Come già fatto per la sequenza possiamo implementare uno stack basandoci su un array.

Possiamo però osservare anche che lo stack può essere considerato come una sequenza con particolari restrizioni d'uso.

Per cui definiremo due classi di implementazione di stack, la prima StackArray basata sull'uso di un array e la seconda StackSequenza basata sull'uso di una sequenza. Sarà interessante confrontare sie le prestazioni sia il tempo e sforzo necessario per lo sviluppo.

Lista concatenata

Partiamo con una definizione della struttura della lista concatenata:

Una lista concatenata è:

- una lista vuota
- oppure un riferimento ad un oggetto (Nodo) che contiene un dato e una lista concatenata

più precisamente è una definizione mutamente ricosiva di Lista e Nodo, in cui la lista si riferisce a nodo e viceversa.

Una implementazione può essere:

```
public class ListaConcatenata<E> {
  Nodo testa;

  private class Nodo<E>{
    E dato;
    ListaConcatenata prossimo;
  }

}
```

Iniziamo con il prevedere un costruttore che implementa la prima parte della definizione, cioè la lista vuota. Aggiungo anche una ridefinizione di toString(), comoda per seguire lo sviluppo.

```
public class ListaConcatenata<E> {
  Nodo testa;

  public ListaConcatenata() {
    testa=null;
  }

  @Override
  public String toString() {
    return "ListaConcatenata{" + "testa=" + testa + '}';
  }

  private class Nodo<E>{
    E dato;
    ListaConcatenata prossimo;

  }

}
```

Vediamo un semplice metodo di test:

```
  public static void testListaConcatenata() {
    ListaConcatenata<String> l =new ListaConcatenata<String>();
    System.out.println("l="+l);
  }
```

La sua esecuzione produce:

```
l=ListaConcatenata{testa=null}
```

Ora passiamo a implementare la seconda parte della definizione, cioè il Nodo:

```
  private class Nodo<E>{
    E dato;
    ListaConcatenata prossimo;

    public Nodo(E dato, ListaConcatenata prossimo) {
      this.dato = dato;
      this.prossimo = prossimo;
    }

    public E getDato() {
      return dato;
    }

    public void setDato(E dato) {
      this.dato = dato;
    }

    public ListaConcatenata getProssimo() {
      return prossimo;
    }

    public void setProssimo(ListaConcatenata prossimo) {
      this.prossimo = prossimo;
    }

    @Override
    public String toString() {
      return "Nodo{" + "dato=" + dato + ", prossimo=" + prossimo +
'}';
    }

  }
```

Per rendere utilizzabile la ListaConcatenaa serve un metodo per inserire un dato:

Metodi per la lista:

- primo() restituisce il primo dato della lista
- resto() restituisce la lista che segue il primo dato

cc