Unità 13

Tipi astratti di dato e loro realizzazione in Java

Sommario

- Tipo astratto di dato
- Specifica di tipi astratti
- Astrazione di valori e astrazione di entità
- Realizzazione di tipi astratti in Java
- Scelta della rappresentazione dei valori
- Scelta dello schema realizzativo
- Interferenza e sharing
- Scelta dell'interfaccia della classe
- Realizzazione del corpo dei metodi
- Uguaglianza
- Realizzazione di classi astrazione di valori semplici
- Realizzazione di classi astrazione di valori collezione
- Realizzazione di classi astrazione di entità

13.1 Introduzione

Abbiamo visto che uno degli aspetti più importanti nella programmazione è l'astrazione sugli oggetti. In particolare il progettista software individua quali sono gli oggetti del dominio di interesse, li raggruppa concettualmente in classi, individua relazioni tra classi e stabilisce quali sono le proprietà degli oggetti di ciascuna classe.

Per cogliere esattamente cosa una data classe concettuale descrive si utilizza tipicamente la nozione di **tipo astratto di dato**.

L'interpretazione intuitiva di tipo di dato è quella che assegna a tale termine il significato di classe di oggetti su cui è possibile eseguire un insieme prefissato di operazioni.

Nota: si utilizza il termine dato come **sinonimo** del termine oggetto del dominio d'interesse.

13.2 Tipo astratto di dato

In prima approssimazione, un tipo astratto di dato è un **oggetto matematico** costituito da:

- un insieme, detto il **dominio del tipo**, che raccoglie i valori del tipo;
- un insieme di **funzioni**, anche dette **operazioni**, che corrispondono alle operazioni che si possono effettuare sui valori del tipo.

Esempio: Il tipo di dato astratto *Boolean* è definito da:

- Dominio: {*true, false*}, cioè i due valori di verità "vero" e "falso";
- Funzioni: *"or", "and", "not"*, con il significato usuale.

Un tipo astratto di dato permette di specificare dati (oggetti) in modo astratto, indipendentemente da un particolare linguaggio di programmazione. In particolare, un tipo astratto di dato è indipendente da come i dati stessi sono rappresentati ed è indipendente da come sono realizzate le funzioni che operano su tale rappresentazione dei dati.

13.3 Un esempio di tipo astratto di dato

Un esempio più complesso è il tipo astratto *Insieme*, in cui il dominio, che chiameremo *Ins*, è formato da insiemi di elementi di un certo dominio T, e le operazioni sono:

- *insiemeVuoto*, che restituisce il valore corrispondente all'insieme vuoto.
- *estVuoto(i)*, che restituisce true se l'insieme i è il valore corrispondente all'insieme vuoto, false altrimenti.
- *inserisci(i,e)*, che restituisce l'insieme j ottenuto dall'insieme i aggiungendo l'elemento e; se e appartiene già a i allora j coincide con i.
- *elimina(i,e)*, che restituisce l'insieme j ottenuto dall'insieme i eliminando l'elemento e; se e non appartiene a i allora j coincide con i.
- *membro(i,e)*, che restituisce true se l'elemento e appartiene all'insieme i, false altrimenti.
- *scegli(i)*, che restituisce un elemento (qualsiasi) di i, ed è applicabile solo se l'insieme i non è l'insieme vuoto.

Nota:

- In genere un tipo astratto è costituito da più domini, oltre al dominio del tipo (il cosiddetto dominio di interesse). Nel caso dell'insieme, i domini in gioco sono: il dominio di interesse *Ins*, il dominio del tipo *Boolean*, e il dominio T degli elementi che formano gli insieme;
- Le funzioni possono essere definite su diversi domini, ossia la loro applicazione può richiedere zero, uno o più argomenti, e restituiscono un valore di un certo dominio. Inoltre, la definizione delle funzioni richiede una descrizione della legge con cui la funzione calcola il risultato in dipendenza degli argomenti.

13.4 Definizione matematica di tipo astratto di dato

La seguente definizione coglie precisamente la nozione di tipo astratto:

> *Un tipo astratto di dato è un oggetto matematico costituito da due componenti:*
>
> 1. *una collezione di domini, uno dei quali, chiamato dominio di interesse, riveste particolare importanza, perché è il dominio del tipo, cioè rappresenta tutti i valori del tipo;*
> 2. *un insieme di funzioni, ciascuna delle quali ha come dominio di definizione il prodotto cartesiano di n domini che appartengono alla collezione di cui al punto 1, e come codominio un dominio che appartiene alla collezione stessa.*

Questa definizione evidenzia che, dal punto di vista matematico, un tipo astratto di dato può essere considerato come un'**algebra eterogenea**, cioè un'algebra che coinvolge più insiemi.

13.5 Utilizzi dei tipi astratti

La nozione di tipo astratto consente di concettualizzare:

- i tipi di dato utilizzati nella progettazione di algoritmi e nella realizzazione dei corrispondenti programmi, quali array, liste, insiemi, pile, code, alberi, grafi ecc.; si usa spesso il termine *struttura di dati* per riferirsi a questi tipi di dato.
- classi di oggetti qualsiasi, che si ritengono importanti in una applicazione.

È importante comprendere che un progettista software, non è solo interessato a utilizzare tipi di dato consolidati e ampiamente studiati per realizzare programmi, ma anche a ideare nuovi tipi astratti per rappresentare attraverso essi il dominio di interesse di una applicazione.

Consideriamo, ad esempio, una applicazione in cui è rilevante la classe degli studenti iscritti ad un corso universitario. Possiamo pensare a questa classe come ad un tipo astratto che ha come dominio l'insieme degli studenti e come operazioni fondamentali, ad esempio, l'assegnazione di un numero di matricola, l'iscrizione ad un certo anno di corso, la scelta di un piano di studi, e così via.

13.6 Specifica di tipi astratti

Per fornire la specifica di un tipo astratto adottiamo un metodo basato sull'uso schematico del linguaggio naturale. Tale metodo pur non essendo propriamente formale, ci permette di descrivere esattamente, senza ambiguità, tipi astratti.

TipoAstratto ***T***

Domini

D1 : descrizione del dominio *D1*

...

Dm : descrizione del dominio *Dm*

Funzioni

F1 : descrizione della funzione *F1*

...

Fn : descrizione della funzione *Fn*

FineTipoAstratto

La descrizione del dominio è una descrizione sintetica in linguaggio naturale del dominio, invece la descrizione delle funzioni è più articolata (vedi esempio seguente).

Esempio: Diamo la specifica semiformale del tipo astratto di dati insieme che abbiamo già introdotto.

TipoAstratto ***Insieme(T)***

Domini

Ins : dominio di interesse

T : dominio degli elementi dell'insieme

Funzioni

insiemeVuoto() $\mapsto$ ***Ins***
pre: nessuna
post: RESULT è l'insieme vuoto

estVuoto(Ins i) $\mapsto$ ***Boolean***
pre: nessuna
post: RESULT è true se i è il valore corrispondente all'insieme vuoto, false altrimenti

inserisci(Ins i, T e) $\mapsto$ ***Ins***
pre: nessuna
post: RESULT è l'insieme ottenuto dall'insieme i aggiungendo l'elemento e; se e appartiene già a i allora j coincide con i

elimina(Ins i, T e) $\mapsto$ ***Ins***
pre: nessuna
post: RESULT è l'insieme ottenuto dall'insieme i eliminando l'elemento e; se e non appartiene a i allora RESULT coincide con i

membro(Ins i, T e) $\mapsto$ ***Boolean***
pre: nessuna
post: RESULT è true se l'elemento e appartiene all'insieme i, false altrimenti

scegli(Ins i) $\mapsto$ ***T***
pre: l'insieme i non è l'insieme vuoto
post: RESULT è un elemento (qualsiasi) di i

FineTipoAstratto

Commentiamo il meccanismo di specifica facendo riferimento all'esempio del tipo *Insieme*.

- Dopo la parola **TipoAstratto** compare *Insieme*, ovvero il nome del tipo che si sta definendo, seguito da *(T)*. T rappresenta il dominio degli elementi dell'insieme. Questo dominio non è ulteriormente specificato. Questo vuol dire che il tipo *Insieme* è in realta un tipo **parametrico** rispetto al dominio degli elementi. In fase di utilizzo del tipo astratto *Insieme*, T verrà istanziato ad un particolare dominio, ad esempio il dominio dei valori interi, dei caratteri, o altro.

- Di ciascun dominio si forniscono il nome ed una breve descrizione del dominio che esso rappresenta. Nel caso di *Insieme*, vengono descritti il dominio di interesse *Ins*, che rappresenta l'insieme dei valori del tipo, e il dominio T degli elementi dell'insieme.
- Per brevità, quando questo non induce confusione, evitiamo di inserire nella specifica dei domini di un tipo astratto *T1* il dominio di interesse di un altro tipo astratto *T2*. Inoltre se questo non induce confusione possiamo usare nella specifica delle precondizioni e postcondizioni delle funzioni di *T1* le funzioni definite per *T2*. Ad esempio la specifica di alcune funzioni di *Insieme* fa uso di *Boolean* implicito nel riferimento al nome del tipo, tuttavia non abbiamo inserito (il dominio di interesse di) *Boolean* tra i domini di *Insieme*.

Vengono poi descritte le caratteristiche delle varie funzioni associate al tipo astratto. Esse rappresentano tutte e sole le operazioni primitive (funzioni primitive) con cui si può operare sul tipo. Ogni altra operazione sul tipo deve essere definita in termini delle funzioni primitive. Ogni funzione primitiva viene specificata attraverso le seguenti proprietà:

- La cosiddetta **intestazione** (o segnatura della funzione matematica), ossia una espressione della forma

  $$f(D1\ d1,...,Dn\ dn) \mapsto Dris$$

  dove f è il nome della funzione, n è il numero di argomenti, *D1,...,Dn* sono i domini a cui appartengono rispettivamente gli argomenti *d1,...,dn* di ogni applicazione della funzione (il primo argomento appartiene a *D1*, il secondo a *D2*, e così via), e infine *Dris* è il dominio a cui appartiene il risultato della funzione. Si noti che n può anche essere uguale a 0, nel qual caso la funzione restituisce sempre lo stesso valore; ovvero, la funzione è di fatto una costante. Ad esempio, *InsVuoto* è una costante del dominio *Ins*, in particolare la costante che rappresenta il valore di insieme vuoto.
- Le precondizioni e le postcondizioni, che stabiliscono le condizioni che devono essere verificate rispettivamente prima e dopo ogni applicazione della funzione.
  - Le **precondizioni** determinano le condizioni di applicabilità della funzione. Ad esempio, la precondizione su *scegli* stabilisce che per applicare la funzione *scegli* su un insieme i è necessario che i non sia l'insieme vuoto.
  - Le **postcondizioni** servono di fatto a specificare il significato della funzione, perché stabiliscono le condizioni che devono essere verificate dopo ogni applicazione. Utilizziamo il simbolo convenzionale RESULT per denotare il risultato della funzione in dette condizioni. Ad esempio, il significato della funzione *inserisci* è determinato dalla postcondizione che il risultato RESULT della funzione *inserisci* applicata all'insieme i e all'elemento e è l'insieme ottenuto da i aggiungendo e.

13.7 Astrazione di valori e astrazione di entità

I tipi astratti vengono generalmente classificati secondo due categorie fondamentali, in base al tipo di astrazione alla base della loro definizione:

- tipi come astrazione di valori
- tipi come astrazione di entità.

13.8 Tipi astrazione di valori

I tipi che sono **astrazione di valori** rappresentano elementi che sono da considerare veri e propri valori, manipolabili mediante funzioni che operano su di essi per calcolare altri valori. Un esempio di questa categoria di tipi astratti è il tipo *Boolean*, a cui abbiamo fatto riferimento precedentemente, i cui valori sono appunto valori di verità e le cui funzioni sono *and*, *or*, *not*, ecc, ciascuna delle quali calcola un valore di verità.

Un altro semplice esempio sono i numeri complessi, il cui tipo astratto può essere definito come segue:

TipoAstratto ***NumeroComplesso***

Domini

C : dominio dei complessi – dominio di interesse

R : dominio degli reali

Funzioni

creaComplesso(R r, R i) $\mapsto$ ***C***

pre: nessuna

post: RESULT è il numero complesso avente r come parte reale e i come parte immaginaria

reale(C c) $\mapsto$ ***R***

pre: nessuna

post: RESULT è il valore della parte reale del numero complesso c

immaginaria(C c) $\mapsto$ ***R***

pre: nessuna

post: RESULT è il valore della parte immaginaria del numero complesso c

modulo(C c) $\mapsto$ ***R***

pre: nessuna

post: RESULT è il modulo del numero complesso c

fase(C c) $\mapsto$ ***R***

pre: nessuna

post: RESULT è la fase del numero complesso c

FineTipoAstratto

È importante osservare che anche il tipo *Insieme(T)* è un'astrazione di valori, in particolare è un'astrazione di valori **collezione**. In generale i tipi astrazione di valori collezione caratterizzano collezioni di elementi di un qualche (altro) tipo, strutturato secondo un determinato principio e gestito secondo determinate strategie (vedi dopo).

13.9 Tipi astrazione di entità

I tipi che sono **astrazione di entità** rappresentano elementi, detti appunto entità, che sono caratterizzati da uno **stato** che rappresenta l'insieme delle loro proprietà; possono cambiare il loro stato a fronte dell'esecuzione di opportune operazioni. Un esempio di tipo astrazione di entità è *Persona*, il cui stato è costituito dal nome e dalla residenza.

TipoAstratto ***Persona***

Domini

Persona : dominio delle persone – dominio di interesse

Funzioni

creaPersona(String n, String r) $\mapsto$ ***Persona***
pre: nessuna
post: RESULT è una persona che ha nome n e residenza r

nome(Persona p) $\mapsto$ ***String***
pre: nessuna
post: RESULT è il nome della persona p

residenza(Persona p) $\mapsto$ ***String***
pre: nessuna
post: RESULT è la residenza della persona p

cambiaResidenza(Persona p, String r) $\mapsto$ ***Persona***
pre: nessuna
post: RESULT è la persona p con la residenza cambiata in r

FineTipoAstratto

Il concetto di stato è quello che differenzia le entità dai valori. Ad esempio, un numero complesso ha la parte immaginaria e la parte reale come proprietà, ma non è ragionevole considerarle come lo stato del numero. Infatti non è ragionevole pensare che un numero complesso cambi, ad esempio, la parte immaginaria da 7 a -5: due numeri complessi che hanno 7 e -5 come parti immaginarie rappresentano semplicemente due valori diversi. Al contrario, una persona che ha cambiato residenza può essere considerata un'entità che, a fronte di un'operazione specifica, ha cambiato il suo stato.

Si noti che il concetto di stato nei tipi astrazione d'entità, rimane comunque implicito nella specifica del tipo astratto stesso. Tale nozione emerge solo nella descrizione delle funzioni del tipo.

I tipi che sono astrazione di entità sono anche caratterizzati dal fatto che entità con uguali proprietà non sono necessariamente uguali, poiché ogni entità ha un'identità propria che non coincide semplicemente con l'insieme delle sue proprietà. Ad esempio, due persone con lo stesso nome e la stessa residenza sono comunque due diverse persone, sebbene con uguale stato (vedi dopo paragrafo sull'uguaglianza).

13.10 I tipi astratti come enti matematici

Si noti che la specifica di un tipo astratto, sia esso astrazione di valori o astrazione d'entità, non si riferisce in alcun modo né alla rappresentazione dei valori del dominio d'interesse del tipo, né al fatto che tali valori saranno poi eventualmente memorizzati nelle variabili del programma.

Le operazioni associate al tipo vengono descritte semplicemente come funzioni matematiche che calcolano valori a fronte di altri valori, e non come meccanismi che modificano variabili. Ad esempio, l'applicazione *Inserisci(i,e)* della funzione *Inserisci* all'insieme *i* e all'elemento *e non deve interpretarsi come l'operazione che modifica l'insieme i aggiungendo ad esso l'elemento e, ma piuttosto come la funzione che calcola il valore del dominio Ins corrispondente all'insieme ottenuto da i aggiungendo e.*

In altre parole, in fase di concettualizzazione, le operazioni associate ad un tipo astratto sono specificate mediante funzioni matematiche. Nella successiva fase di realizzazione si procederà alla scelta di come tradurre le varie operazioni del tipo, e si potrà

quindi realizzare ogni operazione o come un metodo che calcola nuovi valori (in linea con quanto descritto nella specifica), o come un metodo che modifica le variabili che rappresentano gli oggetti sui quali è invocato.

13.11 Realizzazione di tipi astratti in Java

Nel resto dell'unità illustriamo una metodologia per la realizzazione di tipi astratti utilizzando classi Java.

I passi fondamentali per la realizzazione di tipi astratti mediante classi Java sono schematizzabili come segue:

1. Ad ogni tipo astratto viene associata una classe Java. Lo scopo è fornire ai moduli clienti un nuovo tipo di dato che rispecchia fedelmente le caratteristiche del tipo astratto.
2. Va scelta la maniera con cui vengono rappresentati i valori del tipo astratto, definendo come campi privati le strutture necessarie. I servizi che la classe offre ai moduli clienti vanno dichiarati come campi pubblici.
3. Vengono scelte le modalità di realizzazione delle operazioni del tipo astratto (side-effect o funzionale) e di gestione della memoria utilizzata per la rappresentazione dei valori del tipo (senza condivisione o con condivisione). In altri termini, viene scelto il cosiddetto **schema realizzativo** della classe.
4. Viene decisa la maniera di tradurre l'intestazione delle funzioni del tipo astratto nell'intestazione delle corrispondenti funzioni di classe, nonchè di quali metodi ereditati da `Object` fare overriding (è di particolare interesse il metodo `equals`).
5. Vengono scritte le definizioni dei metodi della classe.

Nel seguito approfondiremo i vari aspetti legati alla metodologia qui schematizzata. Uno degli aspetti su cui ci soffermeremo con maggiore dettaglio è la scelta dello schema realizzativo, che costituisce una delle scelte più critiche della realizzazione di una classe. Un altro aspetto, correlato al precedente, che tratteremo in dettaglio è come la distinzione tra astrazione di valori ed astrazione d'entità si riflette nella realizzazione delle classi. Inoltre ci occuperemo di come definire l'uguaglianza tra oggetti della stessa classe e come realizzare tale uguaglianza facendo overriding di `equals` (e del metodo `hashCode` ad esso associato).

13.12 Schema realizzativo: realizzazione delle operazioni

La prima scelta che dobbiamo fare riguarda il modo in cui la classe realizza le operazioni del tipo astratto. In base a questa scelta, classifichiamo gli schemi realizzativi come:

- **con side-effect**, nel caso in cui i metodi della classe operano modifiche su oggetti di moduli clienti;
- **funzionali**, nel caso in cui i metodi della classe calcolano opportuni valori, restituendoli come risultati senza eseguire modifiche di oggetti appartenenti a moduli clienti.

13.13 Realizzazioni con side-effect

Le realizzazioni con side-effect sono quelle in cui i metodi della classe che realizzano le operazioni del tipo astratto eseguono side-effect su oggetti passati in ingresso (oggetto di invocazione e parametri). In queste realizzazioni gli oggetti possono essere modificati

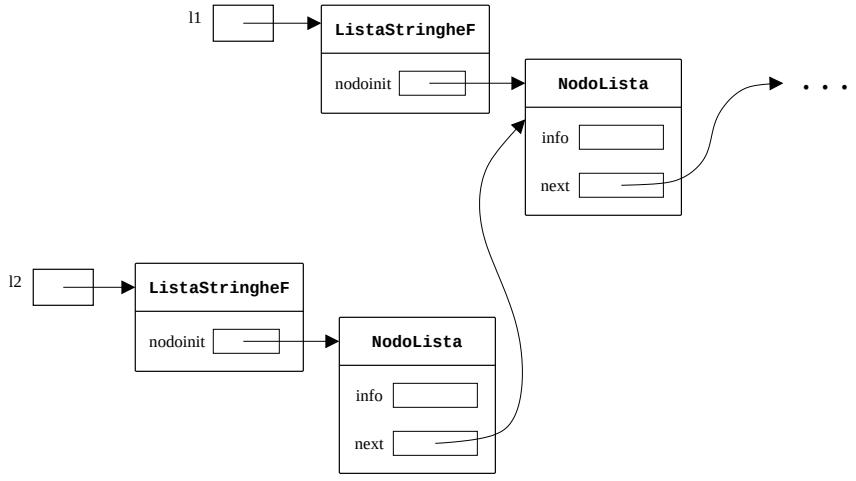
e quindi esse introducono di fatto una nozione di stato dell'oggetto. Tale nozione di stato non è presente esplicitamente nella definizione del tipo astratto e inoltre per quanto riguarda astrazione di valori essa non corrisponde ad una nozione astratta. Ne segue che tali realizzazioni si discostano notevolmente dalla formulazione matematica dei tipi astrazione di valori. Invece nel caso di astrazione di entità l'introduzione della nozione di stato rende le realizzazioni con side-effect particolarmente naturali. In generale le realizzazioni con side-effect portano ad una maggiore efficienza, visto che permettono di operare direttamente sugli oggetti della classe attraverso metodi che effettuano side-effect. Si noti però che proprio la possibilità di fare side-effect richiede di porre particolare attenzione nell'evitare il problema dell'interferenza (cfr. **??**).

13.14 Realizzazioni funzionali

Le realizzazioni funzionali sono quelle in cui i metodi della classe operano concettualmente sui valori del tipo astratto e non eseguono operazioni di modifica di oggetti Java che rappresentano tali valori. La classe viene quindi utilizzata dai moduli clienti per il calcolo dei valori del tipo seguendo molto da vicino la formulazione matematica dei tipi astratti di dato, risultando molto elegante dal punto di vista formale, specie nel caso di tipi che sono astrazione di valori. Le realizzazioni funzionali possono però soffrire di problemi di inefficienza. Infatti, la necessità di restituire sempre nuovi oggetti combinata con il vincolo di non eseguire side-effect può richiedere operazioni di copia (eventualmente parziale) delle strutture, come vedremo per il caso della classe `Insieme`. Inoltre, la scelta di lavorare sui valori, e non sulle variabili, rende meno efficiente la realizzazione in moduli clienti di operazioni non primitive.

13.15 Condivisione di memoria e interferenza

Supponiamo di aver realizzato un classe `ListaStringheF` per rappresentare liste di stringhe, attraverso liste collegate (record-puntatori), adottando uno schema realizzativo con *condivisione di memoria*, con il risultato di condividere alcuni nodi tra più strutture collegate. Ad esempio, consideriamo l'operazione di inserimento in testa: la lista risultante dall'inserimento in una lista di un elemento in prima posizione condivide con la lista originaria tutti gli elementi successivi a quello inserito.



Questa soluzione è efficiente sia in termini di tempo di calcolo che di occupazione di memoria, ma è corretta solamente se lo schema realizzativo adottato è funzionale, cioè non sono presenti metodi che facciano *side-effect*. Infatti se la classe implementasse un metodo con side-effect (**errore gravissimo!**) allora si verificherebbe una *interferenza* tra oggetti distinti della classe.

Esempio: se la classe contenesse un metodo che modifica l'informazione associata al primo elemento della lista

```
public class ListaStringheF {
   ...
   // ERRORE GRAVISSIMO !!!
   public void setFirstInfo(String x) {
      nodoinit.info = x;
   }
}
```

allora il seguente frammento di programma avrebbe effetti collaterali indesiderati.

```
ListaStringheF la = new ListaStringheF().add(0,"A").add(1,"B").
                                                    add(2,"C");
            // la = ( A  B  C )
ListaStringheF lb = la.add(0,"Z");
            // lb = ( Z A B C )
la.setFirstInfo("X"); // ha effetto collaterale su lb !!!
            // la = ( X B C )
            // lb = ( Z X B C )
```

L'effetto descritto si chiama *interferenza*. È un effetto fortemente indesiderato, in quanto modifica il valore di oggetti senza che se ne abbia il controllo, ed è quindi fonte di errori nella programmazione.

Si noti che l'interferenza si può verificare solamente quando concorrono due circostanze:

1. c'è condivisione di memoria;
2. esistono metodi che effettuano side-effect.

13.16 Differenza tra interferenza e sharing

Per utilizzare in modo corretto gli schemi realizzativi occorre distinguere due tipi diversi di condivisione di memoria:

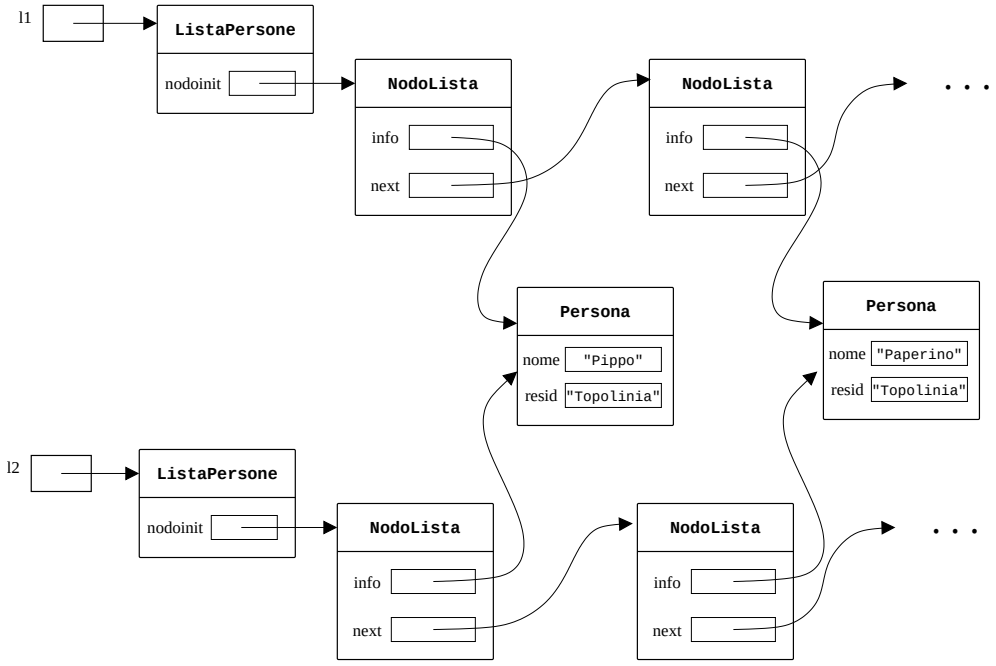
- condivisione di memoria tra gli elementi di strutture collegate (lineare);
- condivisione di memoria per le informazioni associate agli oggetti di interesse per l'applicazione.

La figura nella Sezione **??** illustra un caso di condivisione di memoria tra strutture di una lista, mentre la figura sottostante mostra un caso di condivisione di memoria per rappresentare le informazioni degli oggetti della classe `Persona` (ma non tra i nodi della struttura). La differenza fondamentale tra le due situazioni è che nel primo caso si condivide un elemento della struttura collegata, nel secondo caso l'informazione associata ad un oggetto della classe.

La condivisione di memoria per le informazioni associate agli oggetti di una classe (detta anche *sharing*) è non solo voluta, ma in molti casi addirittura necessaria al fine

di risolvere correttamente il problema in esame. Ad esempio, nella rappresentazione delle due liste nella figura che segue, se la residenza di una persona viene modificata, questo cambiamento è correttamente considerato in entrambe le liste. Se le informazioni associate ad una persona non fossero gestite in *sharing* allora avremmo avuto una inconsistenza nella rappresentazione delle informazioni.

Al contrario la condivisione di memoria degli elementi delle strutture dati atte a rappresentare una collezione, e.g., una lista è una scelta di progetto che deve essere effettuata e realizzata con cura.



13.17 Condivisione di memoria tra oggetti della classe

La seconda scelta che deve essere fatta dal progettista riguarda la modalità con cui la classe gestisce la memoria utilizzata per la rappresentazione dei suoi oggetti. Questa scelta è particolarmente significativa quando il metodo di rappresentazione del tipo astratto prevede di associare agli oggetti della classe strutture di dati realizzate attraverso l'uso di altri oggetti (ad esempio strutture collegate). Le possibilità a disposizione del progettista sono realizzazioni:

- **senza condivisione di memoria**, nel caso in cui i metodi della classe operano in modo da assicurare che le rappresentazioni di oggetti diversi non condividano mai memoria;
- **con condivisione di memoria**, nel caso in cui i metodi della classe non impediscono che le rappresentazioni di oggetti diversi condividano memoria.

Nelle realizzazioni senza condivisione la definizione della classe deve imporre che non ci possa essere condivisione di memoria tra le strutture di dati degli oggetti della classe presenti nel programma. Al contrario, nelle realizzazioni con condivisione, i metodi della classe possono creare condivisione di memoria tra le strutture che fanno parte

di diverse istanze della classe. Esiste cioè la possibilità che due o più oggetti distinti siano rappresentati mediante strutture di dati (ad esempio strutture collegate) che consentano di raggiungere una stessa locazione fisica (ad esempio lo stesso nodo). È evidente che i metodi che modificano tali oggetti debbano essere realizzati con particolare attenzione, affinché non sia compromessa la correttezza dei risultati a fronte di interferenze (cfr. **??**). È altrettanto evidente che, in linea di principio, la condivisione può portare notevoli benefici dal punto di vista dell'efficienza rispetto all'occupazione della memoria.

13.18 Schemi realizzativi privilegiati

In linea di principio, le scelte possibili tra realizzazioni funzionali o con side-effect e con o senza condivisione di memoria portano a quattro schemi realizzativi distinti:

- side-effect, senza condivisione di memoria;
- funzionale, senza condivisione di memoria;
- side-effect, con condivisione di memoria;
- funzionale, con condivisione di memoria.

In realtà in Java il recupero automatico della memoria dinamica non più referenziata favorisce l'uso di due schemi realizzativi:

- lo schema con side-effect senza condivisione di memoria, per realizzare oggetti **mutabili**, cioè appartenenti a classi che sono dotate di metodi che effettuano side-effect cambiandone lo stato;
- lo schema funzionale con condivisione di memoria, per realizzare oggetti **immutabili**, cioè appartenenti a classi che non sono dotate di metodi che effettuano side-effect.

Gli schemi con side-effect senza condivisione sono da preferire agli schemi con side-effect con condivisione, poiché questi ultimi hanno lo svantaggio di richiedere una complessa gestione della memoria, al fine di evitare problemi di interferenza, a cui, in Java, non corrisponde un significativo miglioramento nell'uso della memoria, non essendo possibile controllare con precisione il rilascio della stessa, essendo questo gestito in modo automatico dal garbage collector della Java Virtual Machine.

Per quanto riguarda le realizzazioni funzionali, in Java, sono praticamente sempre preferibili le realizzazioni funzionali con condivisione alle realizzazioni funzionali senza condivisione. Infatti lo svantaggio principale delle realizzazioni con condivisione di memoria è che è difficile stabilire se deallocando un oggetto le sue strutture dati vadano anch'esse deallocate o vadano conservate perchè referenziate da qualche altro oggetto. In Java la gestione automatica della memoria dinamica risolve completamente tale problema.

13.19 Classi come astrazione di valori e astrazione di entità

Come abbiamo visto i tipi astratti si distinguono in:

- tipi astrazione di valori, a loro volta suddivisi
  - tipi astrazione di valori semplici
  - tipi astrazione di valori collezione
- tipi astrazione di entità.

Tale distinzione si riflette nella realizzazione di tipi astratti come classi Java.

13.20 Classi come astrazione di valori semplici

Gli oggetti delle classi che sono **astrazione di valori semplici** rappresentano valori atomici, che non sono cioè considerati collezioni di altri elementi. Per queste classi, il metodo di rappresentazione dei valori del tipo è spesso elementare ed immediato. Il fatto che gli oggetti non rappresentino collezioni di altri elementi implica che non è necessario ricorrere a strutture di dati ausiliarie.

Lo schema realizzativo tipicamente utilizzato di queste classi è quello funzionale. Rispetto alla condivisione di memoria, si noti che se sono sufficienti campi dati (variabili di istanza) di tipi primitivi per la rappresentazione dei valori, allora automaticamente lo schema adottato risulta essere senza condivisione di memoria. Se invece si fa uso di oggetti allora i campi dati conterranno riferimenti a questi, quindi lo schema realizzativo non risulta essere automaticamente senza condivisione di memoria. In queste circostanze, considerando che stiamo rappresentando valori, cioè oggetti immutabili, tipicamente si preferisce uno schema realizzativo con condivisione di memoria.

13.21 Esempio: classe astrazione di valori semplici

Consideriamo, come esempio di questo tipo di classe, la realizzazione del tipo astratto *NumeroComplesso*. Una classe `NumeroComplesso` che realizza il tipo astratto può essere basata sulle seguenti scelte:

1. La rappresentazione di ogni valore del tipo si basa su due campi di tipo `double` nella parte privata: il campo `re` per la parte reale del numero, e il campo `im` per la parte immaginaria.
2. Lo schema realizzativo è (funzionale) senza condivisione di memoria, poiché non vengono utilizzati riferimenti per la rappresentazione dei valori del tipo.

```
public class NumeroComplesso {

  // rappresentazione degli oggetti

  private double re;  // parte reale
  private double im;  // parte immaginaria

  // realizzazione delle funzioni del tipo astratto

  public NumeroComplesso(double r, double i) {
    re = r;
    im = i;
  }

  public double reale() {
    return re;
  }

  public double immaginaria() {
    return im;
  }

  public double modulo() {
    return Math.sqrt(re * re + im * im);
  }
```

```
  public double fase() {
    if (im >= 0) return Math.acos(re/modulo());
    else return -Math.acos(re/modulo());
  }
}
```

13.22 Classi come astrazione di valori collezione

Nelle classi come **astrazione di valori collezione** ogni istanza della classe che rappresenta il tipo astratto è una collezione di istanze di un altro tipo. In questo caso è cruciale scegliere accuratamente il metodo di rappresentazione dei valori del tipo. Il fatto che gli oggetti rappresentano collezioni di altri elementi implica che è necessario ricorrere a strutture di dati eventualmente realizzate con classi ausiliarie.

Ovviamente, le caratteristiche peculiari del tipo astratto sono importanti per la scelta del metodo di rappresentazione dei valori del tipo e dello schema realizzativo. Al contrario del caso di classi come astrazione di valori semplici, non si preferisce necessariamente uno schema realizzativo funzionale, anzi in certe occasioni uno schema realizzativo con side-effect può risultare più adeguato. Per effettuare questa scelta, si possono seguire alcune indicazioni generali:

1. Nel caso in cui i valori del tipo astratto si possano combinare tra loro mediante le funzioni, e nel caso in cui abbia senso costruire espressioni del tipo utilizzando i valori e le funzioni, tipicamente si dimostra più idoneo uno schema funzionale con condivisione, considerando che stiamo trattando valori, cioè oggetti immutabili.
2. Nel caso in cui gli oggetti della classe siano utilizzati singolarmente, cioè indipendentemente l'uno dall'altro si dimostra più idoneo uno schema realizzativo con side-effect senza condivisione di memoria che rende la realizzazione più efficiente. Si noti che in tal caso, le funzioni del tipo astratto sono considerate come delle operazioni che modificano lo stato di un oggetto, come nell'astrazione di entità.

Vedremo nel seguito un esempio dettagliato di realizzazione di una classe come astrazione di valori collezione: la classe `Insieme`.

13.23 Classi come astrazione di entità

La visione di classi come astrazione di entità è quella che più si avvicina alla filosofia della programmazione a oggetti. Tali classi realizzano tipi astratti i cui elementi rappresentano oggetti del mondo reale, detti appunto entità. La caratteristica peculiare delle entità è che ad esse è associato uno stato, cioè un insieme di proprietà, che può cambiare a seguito delle operazioni previste dal tipo astratto. Ne segue che gli oggetti delle classi che realizzano questo genere di tipi astratti sono anch'essi dotati di uno stato che è possibile cambiare attraverso le operazioni previste dalla classe, le quali in ogni caso non cambiano l'identità dell'oggetto che rimane immutata durante tutto il suo ciclo di vita. Il metodo di rappresentazione utilizzato per queste classi è spesso semplice, e consiste nel definire opportuni campi della classe mediante i quali si possano memorizzare le proprietà che caratterizzano lo stato dell'oggetto. Poiché lo stato dell'oggetto cambia durante l'esecuzione del programma, lo schema realizzativo tipicamente adottato è con side-effect senza condivisione. Inoltre, spesso, tali oggetti sono caratterizzati da un **identificatore** che consiste nell'indirizzo dell'oggetto stesso (il riferimento restituito da `new` all'atto della creazione dell'oggetto).

Particolare rilevanza nella realizzazione di classi astrazione d'entità viene assunta dalla necessità di modellare che un oggetto è in relazione con molti oggetti di un dato tipo (relazione **has-many**). Ad esempio una persona può essere proprietario di molti appartamenti. Per rappresentare una tale relazione si deve fare uso di una collezione. Tipicamente tale collezione è realizzata usando uno schema realizzativo con side-effect senza condivisione, poiché viene utilizzata in modo isolato (cfr. paragrafo **??**). Si noti che tale collezione fa concettualmente parte dell'oggetto e quindi, coerentemente con la scelta dello schema realizzativo con side-effect e senza condivisione di memoria, non può essere condivisa con altri oggetti. Quindi bisogna porre particolare attenzione a non introdurre inavvertitamente fenomeni di interferenza sulle strutture dati che reppresentano dette collezioni.

13.24 Uguaglianza

Tutti gli oggetti in Java, qualunque sia la classe a cui appartengono, sono anche implicitamente istanze della classe predefinita `Object`. Quest'ultima è la classe più generale in assoluto: ogni altra classe è implicitamente derivata, direttamente o indirettamente, dalla classe `Object`. Da questo segue che ogni classe eredita le funzionalità di `Object`. Una delle funzionalità più importanti offerte da questa classe è il metodo:

- `public boolean equals(Object ogg)`. Questo metodo verifica l'uguaglianza tra l'oggetto di invocazione e l'oggetto denotato da `ogg`. Per default `equals` considera un oggetto uguale solo a se stesso, cioè si comporta esattamente come "`==`", verificando se i due riferimenti agli oggetti sono uguali, cioè denotano lo stesso oggetto. Questo tipo di uguaglianza è detta **uguaglianza superficiale**. Tuttavia si può fare overriding del metodo `equals` nelle singole classi, facendo in modo che esso restituisca `true` se i due oggetti rappresentano lo stesso **valore** (anche se sono due oggetti distinti) e `false` altrimenti. Questo tipo di uguaglianza, che tiene conto dell'informazione contenuta nell'oggetto, viene detta **uguaglianza profonda** e si contrappone all'uguaglianza superficiale.

Supponiamo di voler ridefinire l'uguaglianza tra oggetti della classe `NumeroComplesso`, considerando uguali oggetti (anche distinti) che rappresentano numeri complessi con la stessa parte reale e la stessa parte immaginaria.

La maniera più semplice di fare ciò è introdurre nella classe la seguente funzione pubblica.

```
public boolean equals(NumeroComplesso c) {
  return re == c.re && im == c.im;
}
```

Si noti che in questo modo abbiamo fatto overloading, ma non overriding, della funzione `equals` in `Object`, in quanto le due funzioni hanno argomenti di tipo diverso. Quest'ultima resta accessibile e può portare a dei risultati controintuitivi. Si consideri ad esempio il seguente programma.

```
public class ProvaEquals {
  static void stampaUguali(Object o1, Object o2) {
    if (o1.equals(o2))
      System.out.println("I DUE OGGETTI SONO UGUALI");
    else
      System.out.println("I DUE OGGETTI SONO DIVERSI");
```

```
  }

  public static void main(String[] args) {
    NumeroComplesso n1 = new NumeroComplesso(10,10);
    NumeroComplesso n2 = new NumeroComplesso(10,10);
    if (n1 == n2)
      System.out.println("I DUE OGGETTI SONO LO STESSO OGGETTO");
    else
      System.out.println("I DUE OGGETTI SONO DISTINTI");
    if (n1.equals(n2))
      System.out.println("I DUE OGGETTI SONO UGUALI");
    else
      System.out.println("I DUE OGGETTI SONO DIVERSI");
    stampaUguali(n1, n2);
  }
}
```

Questo programma stampa:

```
I DUE OGGETTI SONO DISTINTI
I DUE OGGETTI SONO UGUALI
I DUE OGGETTI SONO DIVERSI
```

Vediamo il perché di questo risultato controintuitivo. Il programma crea due oggetti distinti di tipo `NumeroComplesso` che devono essere tuttavia considerati uguali. Essendo i due oggetti distinti, essi hanno riferimenti diversi. Quindi verificando l'uguaglianza mediante l'espressione `n1 == n2` risulta che sono effettivamente distinti. L'uguaglianza tra gli oggetti stessi può essere verificata attraverso la funzione `equals(NumeroComplesso)` di `NumeroComplesso`, che verifica l'uguaglianza dei due oggetti denotati da `n1` e `n2` che risultano effettivamente uguali. Tuttavia invocando la funzione `stampaUguali` i due oggetti risultano diversi. Questo effetto è dovuto al fatto che essendo i parametri formali di `stampaUguali` di tipo `Object`, su di essi viene invocata `equals(Object)`, della quale non si è fatto overriding in `NumeroComplesso`.

Il modo corretto di realizzare il test di uguaglianza è quindi fare overriding di `equals` definita in `Object`, così da evitare il problema evidenziato dal precedente programma. Questo va fatto secondo il seguente schema:

```
public class B {
  ...
  public boolean equals(Object o) {
    if (o == null || !getClass().equals(o.getClass()))
       return false;
    else {
      B b = (B)o;
      test d'uguaglianza sui campi dati di B
    }
  }
  ...
}
```

Notiamo i seguenti aspetti.

- `equals` ha in questo caso esattamente gli argomenti e il tipo di ritorno che ha in `Object`, e quindi ne fa overriding.
- Il test di uguaglianza viene effettuato in due parti. Prima si verifica che l'oggetto di invocazione e l'oggetto denotato da `o` siano entrambi della stessa classe. Ciò viene fatto utilizzando il metodo `getClass` definito in `Object`[1] che restituisce la classe dell'oggetto di invocazione (cioè, la classe più specifica di cui l'oggetto d'invocazione è istanza).[2]
- Se l'oggetto di invocazione e l'oggetto denotato da `o` appartengono effettivamente alla stessa classe, allora `o` denota una istanza di tipo `B`, e possiamo quindi definire una variabile `b` di tipo `B` e assegnare a questa l'oggetto denotato da `o` attraverso una conversione di tipo esplicita. A questo punto, possiamo verificare l'uguaglianza tra i singoli campi della classe `B`.

Ad esempio nel caso di `NumeroComplesso`, seguendo lo schema introdotto, definiamo l'uguaglianza come:

```
public boolean equals(Object o) {
  if (o == null || !getClass().equals(o.getClass()))
     return false;
  else {
    NumeroComplesso c = (NumeroComplesso)o;
    return re == c.re && im == c.im;
  }
}
```

Qualora si desideri specializzare il comportamento dell'uguaglianza per una classe `D` derivata da `B`, si può fare overriding di `equals` secondo il seguente schema semplificato:

```
public class D extends B {
  ...
  public boolean equals(Object o) {
    if (!super.equals(o))
      return false;
    else {
      D d = (D)o;
      test d'uguaglianza campi dati specifici di D
    }
  }
  ...
}
```

Qui `super` è un riferimento standard qui usato per denotare il metodo `equals` definito nella classe base `B`.

13.25 Il metodo `hashCode`

La classe `Object` contiene anche il metodo `int hashCode()` che è strettamente correlato al metodo `equals`.

[1] Il metodo `getClass` è dichiarato `final` in `Object` e quindi non se ne può fare overriding.

[2] Più precisamente, `getClass` restituisce un oggetto della classe predefinita `Class` associato alla classe dell'oggetto di invocazione. Si osservi che esiste un oggetto di classe `Class` per ogni classe definita nel programma.

Questo metodo restituisce un `int` che rappresenta un codice hash per l'oggetto.

In particolare Java, per il corretto funzionamento, richiede che il seguente principio sia sempre rispettato:

> *Se due oggetti sono uguali secondo il metodo* `equals`*, allora il metodo* `hashCode` *restituisce per entrambi lo stesso valore.*

Si noti che il viceversa naturalmente non è richiesto: due oggetti con lo stesso `hashCode` possono essere diversi secondo `equals`.

Il metodo `hashCode` in `Object` costruisce il codice hash direttamente sull'indentificatore dell'oggetto. In questo modo rispetta il principio su menzionato: `equals` se non ridefinito restituisce `true` solo se due oggetti sono in realtà lo stesso oggetto, cioè hanno lo stesso identificatore, e in tal caso ovviamente `hashCode` restituisce lo stesso valore come codice hash.

D'altra parte ogni volta che ridefiniamo `equals` dobbiamo ridefinire anche `hashcode` per rispettare il principio sopra riportato.

Ad esempio nel caso di `NumeroComplesso`, seguendo il principio introdotto, possiamo definire `hashCode` come:

```
public int hashCode() {
    return (re + 3*im) % 31;
    o una qualsiasi altra funzione hash anche banale
    sui campi usati in equals.
}
```

Qualora si desideri specializzare il comportamento di `hashCode` per una classe `D` derivata da `B`, si può fare overriding di `hashCode`, seguendo ancora lo schema adottato per `equals`:

```
public class D extends B {
  ...
  public int hashCode() {
    return  codice hash  definito su super.hashCode()
                 e campi dati specifici di D
  }
  ...
}
```

13.26 Uguaglianza nelle classi astrazione di valori

Nelle classi astrazione di valori (sia valori semplici che valori collezione) è sempre opportuno effettuare overriding del metodo `equals` (e `hashCode`) ereditato dalla classe `Object`, facendo in modo che due oggetti siano considerati uguali (da `equals`) quando, anche se distinti, rappresentano lo stesso valore. Ricordiamo che questo tipo di uguaglianza, che tiene conto dell'informazione contenuta nell'oggetto, viene detta **uguaglianza profonda** e si contrappone all'uguaglianza di riferimenti che denotano lo stesso oggetto, ovvero all'uguaglianza superficiale.

13.27 Uguaglianza nelle classi astrazione di entità

Tipicamente nelle classi astrazione di entità il tipo di unguaglianza di interesse è l'**uguaglianza superficiale** che stabilisce che due oggetti sono uguali se e solo se sono in realtà lo stesso oggetto. Infatti due entità sono uguali solo se in realtà sono la stessa entità. Di conseguenza il metodo `equals` (ed `hashCode`) ereditato da `Object` è già adeguato a fare la verifica di uguaglianza, e non occorre farne overriding.

13.28 La progettazione della classe `Insieme`

In questa sezione approfondiremo alcune delle considerazioni fatte sulla realizzazione di tipi astratti, astrazione di valori collezione, mediante classi Java, illustrando la progettazione della classe `Insieme`, che realizza il tipo astratto *Insieme* la cui specifica è stata data all'inizio dell'unità. Presenteremo due realizzazioni: la prima adotta uno schema realizzativo con side-effect senza condivisione, mentre la seconda adotta uno schema funzionale con condivisione.

13.29 `Insieme`: considerazioni generali

Per rappresentare un insieme utilizziamo una lista collegata senza duplicazioni di elementi, cioè ogni elemento dell'insieme occorre esattamente una volta nella lista.

Per definire la lista, definiamo la classe `NodoLista` che contiene al suo interno un campo `info` che è un riferimento ad un oggetto della classe `Object` (l'elemento dell'insieme) ed un campo `next` che è un riferimento ad un altro oggetto della stessa classe `NodoLista` (il resto della lista). Tale classe, essendo solo una struttura ausiliaria per la classe `Insieme`, viene definita nello stesso file della classe `Insieme` ma non viene dichiarata pubblica.

Nella classe `Insieme` definiamo un campo privato `inizio` di tipo `NodoLista`, in modo che ogni oggetto della classe contenga un riferimento al primo elemento di una lista collegata. Il valore `null` di `inizio` indica che l'oggetto rappresenta un insieme vuoto. Si osservi che, poiché questo campo dati è privato, non sarà possibile accedervi direttamente da parte di moduli clienti. Esso potrà essere acceduto solo attraverso i metodi della classe.

La classe `Insieme` che stiamo definendo è costituita da una collezione di `Object` Java, quindi definisce insiemi i cui elementi possono essere oggetti qualsiasi. Possiamo però facilmente modificare la definizione della classe in modo che gli elementi dell'insieme che essa rappresenta siano tutti dello stesso tipo, per esempio stringhe o interi. Sostanzialmente basta modificare il tipo del campo `info` della classe `NodoLista` ed il tipo dei parametri formali che rappresentano elementi dell'insieme nei vari metodi e poco altro.

13.30 `Insieme`: realizzazione con side-effect senza condivisione

La realizzazione con side-effect senza condivisione è analoga alla maggior parte delle realizzazioni di classi effettuate fino ad adesso.

```
class NodoLista {
  Object info;
  NodoLista next;
}
```

```
public class InsiemeSS {

  // rappresentazione dell'oggetto

  private NodoLista inizio;

  // realizzazione delle funzioni del tipo astratto

  public InsiemeSS() { // realizza la funzione InsVuoto
    inizio = null;
  }

  public boolean estVuoto() {
    return inizio == null;
  }

  public boolean membro(Object e) {
    return appartiene(e,inizio);
  }

  public void inserisci(Object e) {
    if (!appartiene(e,inizio)) {
      NodoLista l = new NodoLista();
      l.info = e;
      l.next = inizio;
      inizio = l;
    }
  }

  public void elimina(Object e) {
    inizio = cancella(e,inizio);
  }

  public Object scegli() {
    if (inizio == null) return null;
    else return inizio.info;
  }

  // uguaglianza

  public boolean equals(Object o) {
    if (o == null || !(getClass().equals(o.getClass())))
      return false;
    else {
      InsiemeSS ins = (InsiemeSS)o;
      NodoLista l;
      l = inizio;       // verifica che tutti gli elementi di inizio
                        // siano anche elementi di ins.inizio
      while (l != null) {
        if (!appartiene(l.info,ins.inizio))
          return false;
        l = l.next;
      }
      l = ins.inizio;   // verifica che tutti gli elementi di ins.inizio
                        // siano anche elementi di inizio
```

```
      while (l != null) {
        if (!appartiene(l.info,inizio))
          return false;
        l = l.next;
      }
      return true;
    }
  }

  public int hashCode() {
    return 1;  // ridefina in modo banale ma coerente con equals
  }

  // metodi ausiliari

  private static boolean appartiene(Object e, NodoLista l){
    return (l != null) && (l.info.equals(e) || appartiene(e,l.next));
  }

  private static NodoLista cancella(Object e, NodoLista l) {
    if (l == null) return null;
    else  if (l.info.equals(e)) return l.next;
    else {
      l.next = cancella(e,l.next);
      return l;
    }
  }
}
```

Commentiamo le principali caratteristiche della definizione della classe.

- L'interfaccia di classe è costituita da un metodo pubblico per ogni funzione del tipo astratto, dove l'insieme su cui le funzioni operano è rappresentato dall'oggetto di invocazione (scelta tipica). Si noti che `inserici` ed `elimina` agiscono direttamente sull'oggetto di invocazione effettuando quindi side-effect. La funzione *InsVuoto* del tipo astratto *Insieme* è stata realizzata attraverso il costruttore della classe. Tale costruttore costruisce un oggetto che rappresenta l'insieme vuoto.
- Diversi metodi fanno uso dei due metodi ausiliari private `appartiene` e `cancella`, i quali sono entrambi definiti `static` e quindi sono associati alla classe `InsiemeSS` e non alle singole istanze. Questi metodi infatti operano su oggetti della classe `NodoLista` e non su oggetti della classe `InsiemeSS`. Il metodo `appartiene` verifica se un certo elemento fa parte di una lista di elementi; si noti che per fare ciò utilizza il metodo `equals` sugli elementi della lista. È molto importante notare che nonostante il riferimento `l.info` sia di tipo `Object` il metodo `equals` effettivamente invocato non è quello di `Object`, ma quello ridefinito (overridden) nella classe più specifica a cui appartiene l'oggetto denotato da tale riferimento. Questo è dovuto al meccanismo di late binding adottato da Java. Il metodo `cancella`, utilizzato dal metodo pubblico `elimina`, serve a cancellare un elemento dalla lista.
- Poiché stiamo realizzando un tipo astrazione di valori, facciamo overriding in modo opportuno del metodo `equals` ereditato da `Object` in modo che consideri uguali due oggetti, anche distinti, che rappresentano lo stesso insieme.

- Infine, per brevità si è omesso di fare overriding di `toString`.

13.31 `Insieme`: realizzazione funzionale con condivisione

Nelle realizzazioni funzionali l'assenza di side-effect nelle funzioni che realizzano le operazioni del tipo astratto consente un certo grado di condivisione di memoria senza cadere nel problema dell'interferenza, come mostra la seguente realizzazione.

```
class NodoLista {
  Object info;
  NodoLista next;
}

public class InsiemeFC {

  // rappresentazione dell'oggetto

  private NodoLista inizio;

  // realizzazione delle funzioni del tipo astratto

  public InsiemeFC() { // realizza la funzione insVuoto
    inizio = null;
  }

  public boolean estVuoto() {
    return inizio == null;
  }

  public boolean membro(Object e) {
    return appartiene(e,inizio);
  }

  public InsiemeFC inserisci(Object e) {
    if (appartiene(e,inizio)) return this;
    else {
      InsiemeFC ins = new InsiemeFC();
      ins.inizio = new NodoLista();
      ins.inizio.info = e;
      ins.inizio.next = inizio; // condivisione di memoria
      return ins;
    }
  }

  public InsiemeFC elimina(Object e) {
    if (!appartiene(e,inizio)) return this;
    else {
      InsiemeFC ins = new InsiemeFC();
      ins.inizio = cancella(e,inizio); // effettua anche copia parziale
      return ins;                      // per evitare side-effect
    }
  }

  public Object scegli() {
    if (inizio == null) return null;
```

```
    else return inizio.info;
  }

  // uguaglianza

  public boolean equals(Object o) {
    if (o == null || !(getClass().equals(o.getClass())))
      return false;
    else {
      InsiemeFC ins = (InsiemeFC)o;
      NodoLista l;
      l = inizio;      // verifica che tutti gli elementi di inizio
                       // siano anche elementi di ins.inizio
      while (l != null) {
        if (!appartiene(l.info,ins.inizio))
          return false;
        l = l.next;
      }
      l = ins.inizio;  // verifica che tutti gli elementi di ins.inizio
                       // siano anche elementi di inizio
      while (l != null) {
        if (!appartiene(l.info,inizio))
          return false;
        l = l.next;
      }
      return true;
    }
  }

  public int hashCode() {
    return 1;  // ridefina in modo banale ma coerente con equals
  }

  // metodi ausiliarie

  private static boolean appartiene(Object e, NodoLista l){
    return (l != null) && (l.info.equals(e) || appartiene(e,l.next));
  }

  private static NodoLista cancella(Object e, NodoLista l){
    if (l == null) return null;
    else if (l.info.equals(e)) return l.next;
    else {
      NodoLista ll = new NodoLista();
      ll.info = l.info;
      ll.next = cancella(e,l.next);
      return ll;
    }
  }
}
```

Commentiamo le differenze fondamentali rispetto alla realizzazione precedente.

- Come nella realizzazione precedente, l'interfaccia di classe è costituita da un metodo pubblico per ogni funzione del tipo astratto, dove l'insieme di input per

le funzioni è rappresentato dall'oggetto di invocazione (scelta tipica).

- Abbiamo introdotto due metodi ausiliari private `appartiene` e `cancella` che operano su oggetti della classe `NodoLista`. Il metodo `appartiene` è identico a quello definito nella realizzazione precedente. Il metodo `cancella` che cancella un elemento dalla lista, ed è diverso dal metodo omonimo di `InsiemeSS`. Infatti per evitare il problema dell'interferenza, visto che tale lista potrebbe essere condivisa con qualche altro oggetto di tipo `InsiemeFC`, si comporta in modo particolare: duplica la lista fino al nodo contenente l'elemento da cancellare (escluso) e poi crea condivisione con i nodi che seguono l'elemento da cancellare.

- Per restituire un nuovo insieme, I metodi `inserisci` ed `elimina` creano un oggetto di tipo `InsiemeFC` che viene aggiornato opportunamente e poi restituito come risultato. L'istruzione `return this` viene utilizzata per restituire l'oggetto di invocazione, quando l'inserimento e la cancellazione non devono costruire nuove liste.

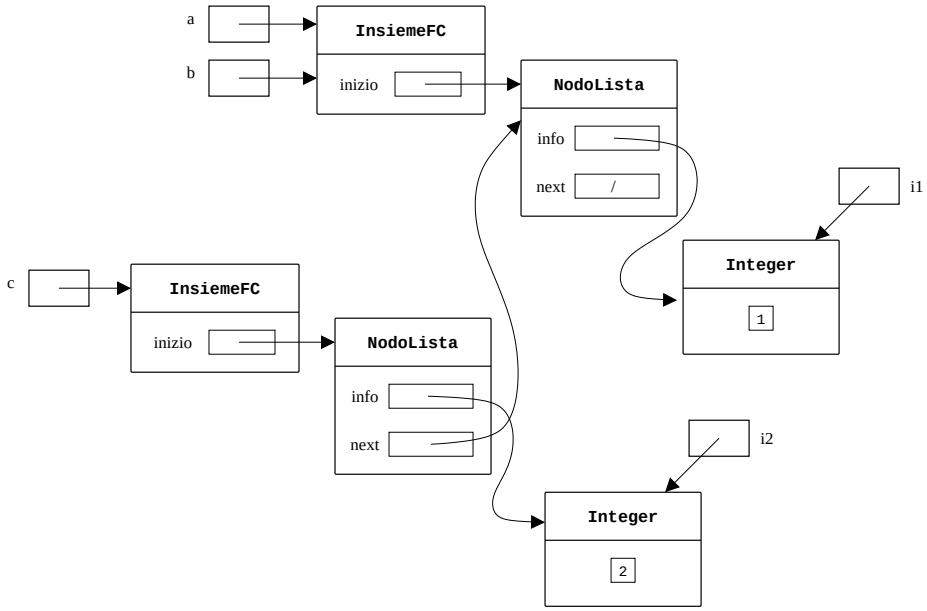
- Poiché stiamo realizzando un tipo astrazione di valori, si è fatto overriding in modo opportuno del metodo `equals` ereditato da `Object` in modo da considerare uguali due oggetti, anche distinti, che rappresentano lo stesso insieme. Tale metodo è identico a quello visto nella realizzazione precedente.

- Per brevità si è omesso di fare overriding di `toString`.

Per chiarire come i metodi della classe gestiscono la condivisione di memoria tra oggetti, si consideri il seguente frammento di codice in cui vengono dichiarati vari riferimenti ad oggetti della classe `InsiemeFC` con elementi della classe predefinita `Integer` che serve ad incapsulare interi in oggetti:

```
InsiemeFC a, b, c;
Integer i1 = new Integer(1);
Integer i2 = new Integer(2);
a = new InsiemeFC();
a = a.inserisci(i1);
b = a;
c = a.inserisci(i2);
```

La figura seguente rappresenta la situazione in memoria dopo l'esecuzione di queste istruzioni. Si noti che l'insieme `c` condivide con gli insiemi `a` e `b` l'elemento `i1`, nel senso che `c.inizio.next` si riferisce esattamente all'oggetto di tipo `NodoLista` a cui si riferisce `a.inizio` (che è uguale a `b.inizio`). Inoltre, l'insieme `c` contiene un riferimento all'elemento `i2`. Si noti che questa condivisione non rappresenta un problema poiché la classe `InsiemeFC` non mette a disposizione dei moduli cliente metodi che effettuano side-effect.



13.32 La progettazione della classe `Studente`

Ci occupiamo ora della realizzazione, mediante una classe Java, del tipo astratto *Studente* definito come segue:

TipoAstratto ***Studente***

Domini

Studente : dominio degli studenti – dominio di interesse

Esame : dominio degli esami

PianoDiStudio : dominio dei piani di studio

Funzioni

creaStudente(Stringa n, Stringa f, PianoDiStudi p) $\mapsto$ ***Studente***

pre: nessuna

post: RESULT è uno studente avente come nome n, iscritto alla facoltà f, avente come piano di studi p e che non ha superato alcun esame

nome(Studente s) $\mapsto$ ***Stringa***

pre: nessuna

post: RESULT è il nome dello studente s

facolta(Studente s) $\mapsto$ ***Stringa***

pre: nessuna

post: RESULT è il la stringa che rappresenta la facoltà a cui lo studente s è iscritto

piano(Studente s) $\mapsto$ ***PianoDiStudi***

pre: nessuna

post: RESULT è il piano di studi dello studente s

assegnaFacolta(Studente s, Stringa f) $\mapsto$ ***Studente***

pre: nessuna

post: RESULT è lo studente s con la facoltà a cui è iscritto modificata in f

media(Studente s) $\mapsto$ ***Reale***
pre: nessuna
post: RESULT è la media dei voti ottenuti dallo studente *s* negli esami sostenuti

votoEsame(Studente s, Esame e) $\mapsto$ ***Intero***
pre: nessuna
post: RESULT è il voto ottenuto dallo studente *s* nell'esame *e*, se effettivamente superato; altirimenti RESULT è pari a 0

FineTipoAstratto

Le informazioni fondamentali associate agli oggetti della classe `Studente` saranno il nome e la facoltà a cui sono iscritti. Inoltre, assoceremo ad ogni studente il proprio piano di studi e gli esami sostenuti con il rispettivo voto (relazione **has-many**). Assumeremo che i piani di studio e gli esami siano rappresentati come istanze di due classi già definite `PianoDiStudi` ed `Esame` rispettivamente.

Essendo il tipo astratto *Studente* una astrazione di entità, nel realizzare la classe corrispondente seguiamo quanto detto nel paragrafo **??**. In particolare, scegliamo lo schema realizzativo con side-effect senza condivisione.

Realizziamo la relazione tra `Studente` e `PianoDiStudi` memorizzando l'identificatore dell'oggetto di tipo `PianoDiStudi` associato ad una data istanza di `Studente`, e decidiamo di porre l'oggetto di tipo `PianoDiStudi` in sharing.

Invece per realizzare la relazione **has-many** tra `Studente` e la coppia esame sostenuto e rispettivo voto dobbiamo fare uso di una collezione. In particolare, realizziamo tale collezione untilizzando un array la cui dimensione massima poniamo a 30 (massimo numero esami). Si noti che in questo modo scegliamo per la collezione una realizzazione con side-effect senza condivisione, visto che non prevediamo di comporre tale collezione con altre collezioni, ma soltanto di accedervi attraverso i metodi della classe `Studente`. Per rappresentare gli elementi di tale insieme facciamo uso di una classe ausiliaria `EsameVoto` avente due funzioni pubbliche `esame` che restituisce l'esame e `voto` che restituisce il voto.

```
class EsameVoto {
  private Esame es;
  private int vo;
  public EsameVoto(Esame e, int v) {
    es = e;
    vo = v;
  }
  public Esame esame() { return es; }
  public int voto() { return vo; }
}

public class Studente {

  // rappresentazione dell'oggetto

  private String nome;
  private String facolta;
  private PianoDiStudi piano_studi;
  private EsameVoto[] esamiFatti;
  private int numEsami;
```

```
 // realizzazione delle funzioni del tipo astratto

  public Studente(String n, String f, PianoDiStudi p) {
                    // realizza la funzione creaStudente
    nome = n;
    facolta = f;
    piano_studi = p;
    esamiFatti = new EsameVoto[30];
    numEsami = 0;
  }

  public String nome() {
    return nome;
  }

  public String facolta() {
    return facolta;
  }

  public PianoDiStudi piano() {
    return piano_studi;
  }

  public void assegnaFacolta(String f) {
    facolta = f;
  }

  public void inserisciEsame(Esame e, int v) {
    esamiFatti[numEsami] = new EsameVoto(e,v);
    numEsami++;
  }

  public double media() {
    double somma = 0;
    for(int i = 0; i < numEsami; i++)
      somma = somma + esamiFatti[i].voto();
    return (double)somma/numEsami;
  }

  public int votoEsame (Esame e) {
    for(int i = 0; i < numEsami; i++)
      if (esamiFatti[i].esame().equals(e)) return esamiFatti[i].voto();
    return 0; // lo studente non ha sostenuto l'esame e
  }

  // uguaglianza
  // public boolean equals(Object) ereditata da Object e' sufficiente
  // e quindi anche int hashCode()
}
```

Commentiamo le caratteristiche principali della definizione della classe.

- Coerentemente con la scelta di usare una rappresentazione con side-effect, le funzioni `assegnaFacolta` e `inserisciEsame` non restituiscono un nuovo oggetto,

ma modificano l'oggetto di invocazione.

- La classe ausiliaria `EsameVoto`, essendo d'interesse solo nell'ambito della classe `Studente` è stata definita come classe non pubblica all'interno del file che contiene `Studente`.
- Per rappresentare la relazione tra `Studente` e `PianoDiStudi` abbiamo utilizzato il campo privato `piano` dove poniamo in sharing l'oggetto di tipo `PianoDiStudi`.
- Per rappresentare la relazione **has-many** tra `Studente` e `EsameVoto` abbiamo utilizzato un campo privato `esamiFatti` di tipo array di (riferimenti a) oggetti di tipo `EsameVoto`. Inoltre per rappresentare quali campi sono significative nell'array (gli esami effettivamente sostenuti) abbiamo utilizzato un campo intero `numEsami`. Coerentemente, nel costruttore questo campo viene inizializzato a 0.
- Non si è fatto overriding di `equals` (e si assume che neanche in `Persona` lo si sia fatto), poiché, essendo `Studente` (come `Persona`) una astrazione di entità, gli oggetti di tipo `Studente` sono uguali solo a se stessi e quindi il comportamento di `equals` definita in `Object` è già adeguato.
- Infine, per brevità, non si riporta il codice di `toString`, di cui si può fare overriding in modo opportuno se necessario.