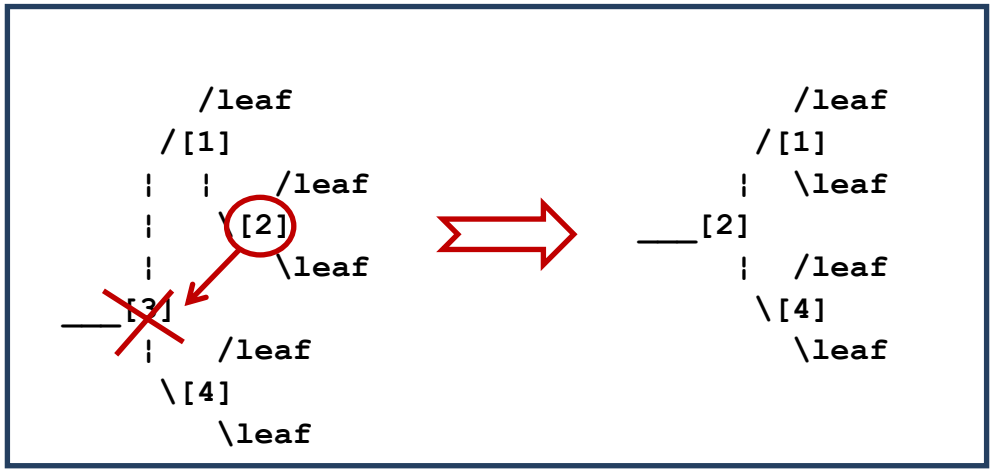
Corso: “Programmazione IIB” Diario delle Lezioni e delle Esercitazioni

Università di Torino
Corso di Laurea in Informatica
A.A. 2018-2019

Docente: Stefano Berardi

A cura di S.Berardi, materiale originario di L. Padovani, con contributi di: D. Magro, G. Torta, M. Baldoni, S. Tedeschi



Rimozione di un nodo da un albero di ricerca

Presentazione del Corso: Programmazione II - Teoria - Corso B
(cognomi L-Z)

Nel 2019 questo corso si svolge in 48 ore (24 lezioni di 2 ore) e vale 6 crediti, è associato a 6 ore di esercitazioni (3 lezioni di 2 ore, su due turni) e 30 ore di Laboratorio (10 lezioni di 3 ore, su due turni). Il corso è la continuazione del corso di ProgIB del 2018: supporremo note le dispense di

ProgIB (per es. su: www.di.unito.it/~stefano/2018-12-19-ProgIB.pdf)

Testo di consultazione.

- Walter Savitch, ***Programmazione di base e avanzata con Java***, Pearson, 2014 (capitoli 8-13, 15-16).

Programma del Corso

Rispetto al libro dedicheremo piu' spazio a: **1.** definizioni ricorsive **2.** l'uso di asserzioni per verificare che un programma si comporti come ci aspettiamo **3.** il disegno di Stack e Heap. ***Tutti e tre i punti corrisponderanno a un esercizio di esame.***

Ometteremo invece la sezione 14 del Savich (input/output). Il **capitolo 15, Strutture Dinamiche**, viene **anticipato e ampliato**.

Organizzazione dei laboratori di ProgII B

Il laboratorio inizia l**a *settimana del 4 marzo*** 2019 ed è così suddiviso:

- Turno **1** ***(Gianluca Torta)*** studenti il cui login di laboratorio termina con una cifra **dispari**
- Turno **2** ***(Diego Magro)*** studenti il cui login di laboratorio termina con una cifra **pari**

La Prof. Valentina Gliozzo tiene un laboratorio di 24 ore per gli studenti del 2017-2018 che non hanno ancora passato ProgII, con un orario scelto per evitare sovrapposizioni con i corsi del secondo anno: ***Mercoledì e Giovedì dalle 16 alle 18 nel laboratorio Turing***, a partire da ***Giovedì 7 Marzo***. (In seguito l'orario potrebbe cambiare a seconda delle esigenze, quindi controllate).

Esercitazioni

Ci sono 6 ore di esercitazioni (3 esercitazioni di 2 ore, su due turni). Non sono in laboratorio, vi viene chiesto di risolvere gli esercizi su carta o uno strumento elettronico portatile.

La prima esercitazione sarà **il Mercoledì 13 marzo**, per le date delle altre due decideremo in seguito.

Distribuzione del materiale didattico

Il materiale didattico è distribuito come segue:

- Gli argomenti per il corso e il laboratorio sono di norma pubblicati all'inizio di ogni settimana
- Il codice sorgente visto a lezione è di norma pubblicato entro termine di ogni settimana
- Le soluzioni delle esercitazioni di laboratorio sono di norma pubblicate entro la settimana successiva a quella del loro svolgimento

Tuttavia, ci riserviamo la possibilità di ***anticiparne o posticiparne*** la pubblicazione per ragioni didattiche.

Editor, Compilatore e Visualizzatore Java

Come Editor e Compilatore Java scegliete quello che preferite: per suggerimenti rimandiamo alla pagina del Corso di Laboratorio IB di quest'anno. Potete anche usare un compilatore online, per esempio:

https://www.compilejava.net/

Vi consigliamo di usare un Java Visualizer, per es.:

https://cscircles.cemc.uwaterloo.ca/java_visualize/

per avere una rappresentazione di Stack e Heap durante l'esecuzione di un programma Java. All'esame vi chiederemo di ***disegnare Stack e Heap a mano*** per un esempio.

Modalità di Esame: Laboratorio e Teoria

Gli studenti devono prima passare una prova pratica di laboratorio che fa da sbarramento all'esame di teoria. Il punteggio di laboratorio è ***0 (bocciato), 1 (promosso), 2 (promosso con lode)***. Una settimana dopo gli studenti devono svolgere l'esame di teoria, con 4 domande ciascuna con il suo punteggio, e risposte scritte su carta. Il voto ottenuto vale sia per teoria che per laboratorio. Chi ha 30 all' esame di teoria ottiene ***30 e lode*** se ha un voto 2 a laboratorio, ottiene 30 se ha un voto 1. Non ci sono altre differenze tra prendere 1 e 2 a laboratorio.

Il laboratorio dura fino a Febbraio dell'anno successivo compreso. Se uno studente partecipa all'esame di teoria e non viene promosso ***non deve comunque ripetere il Laboratorio*** se sostiene teoria ***entro Febbraio dell'anno successivo***. Il laboratorio perde valore e va ripetuto il Giugno dell'anno dopo, quando iniziano gli esami del corso successivo.

Tazzina di caffè espresso

Indice delle Lezioni e Esercitazioni di Programmazione IIB del 2019

Lezione 01 Classi, attributi e metodi pubblici e privati

Presentazione del corso. I primi *30 minuti* della lezione 01 sono dedicati alla presentazione del corso (vedi pagine precedenti).

Lezione 1. Classi, attributi e metodi pubblici. *(60 minuti)* Una classe consiste di oggetti e metodi definiti su questi oggetti. Un oggetto è l'indirizzo di un gruppo di dati più semplici, disposti consecutivamente in memoria. Un esempio di oggetto è un vettore di interi (**ProgIB**,cap.6). Come già sapete, un vettore viene identificato con il suo indirizzo e consiste di un gruppo di interi disposti in modo consecutivo (oltre all'intero che ne rappresenta la lunghezza). A ProgII definiremo nuove classi di oggetti.

Un primo esempio di definizione di classe: la classe Cane. Un oggetto x di classe "Cane" (un "cane" x) è composto di dati più semplici detti ***"attributi"***. Nel caso della classe Cane scegliamo come attributi: una stringa (il ***nome***) seguita da una stringa (la ***razza***) seguita da un intero (gli ***anni***). Nella memoria un oggetto è rappresentato dall'indirizzo del suo primo attributo. Gli attributi di un cane x si indicano con: x.nome, x.razza, x.anni, e più in generale con x.attributo.

Eseguibilità di una classe. Nel corso di **ProgIB** avete scritto solo classi con il main, e avete usato come librerie classi prive di main come la classe Math per costanti e operazioni matematiche (sezione 2.6). Una classe è eseguibile solo se contiene un main, altrimenti può solo essere utilizzata come libreria da altre classi. Se definiamo la classe Cane senza un main, allora Cane **non è eseguibile** da sola, ma puo' venir richiamata da un main oppure da un metodo.

Classi pubbliche e private. Possiamo dichiarare Cane, e una qualunque classe, in due modi. **(1)** Possiamo scrivere **class Cane:** in questo caso Cane è utilizzabile solo da classi nello stesso file. **(2)** Possiamo scrivere invece **public class Cane:** in questo caso la classe può essere usata da classi in altri files, ma deve essere salvata nel file Cane.java. Nelle prime lezioni presentiamo alcune classi non pubbliche come esempio, dopo useremo solo classi pubbliche.

Metodi dinamici pubblici. Nelle dispense di **ProgIB** (cap.3) avete visto i metodi statici. Una classe può contenere metodi statici. Una

classe può anche contenere metodi dinamici pubblici. Un metodo dinamico pubblico si scrive come:

public T metodo(argomenti){… istruzioni …}

Per ora supporremo che non ci siano argomenti, dunque che (argomenti) sia la lista vuota (). Dato un oggetto x di Cane, se scrivo x.metodo() mando il metodo dinamico "metodo" a un cane x e ricevo in risposta un risultato di tipo T (dove T può essere **void**). Possiamo descrivere x come un argomento di metodo(), scritto davanti al nome metodo ("prefisso") anziché dopo il nome metodo, come avviene di solito. All'interno della definizione di ***"metodo"***, quando scrivo ***"nome"***, ***"razza"***, ***"anni"*** intendo nome, razza ed anni del cane x. All'interno di metodo(), il cane x a cui mando il metodo può venire indicato con ***"this"***: in questo caso, ***"this.nome"***, ***"this.razza"***, **"this.anni"** sono: nome, razza ed anni del cane x a cui mando il metodo.

Un esempio di metodo dinamico: i metodi di input. Come esempio, utilizziamo una classe Scanner. Uno scanner x è capace di leggere un input da tastiera e tradurlo nella sua rappresentazione Java. In questo caso abbiamo un metodo dinamico per ogni tipo di input, per esempio se inviamo nextLine() a un scanner x e inseriamo una stringa da tastiera otteniamo in risposta rappresentazione Java di una stringa.

Nota sui caratteri ASCII. Quando scriviamo del codice Java non utilizziamo accenti, perché molti compilatori non li leggono. Utilizziamo al loro posto gli apostrofi: scriveremo ***a',e',e',i',o',u'*** al posto di ***à,è,é,ì,ò,ù***. Per la stessa ragione scriveremo '' e "" al posto di ‘’ e di "".

```
//Salveremo il tutto nel file: CaneDemo.java
import java.util.Scanner; //Usiamo la classe Scanner (java utility)

class Cane // Classe non pubblica
//utilizzabile solo all'interno dello stesso file
{
    public String nome;
    public String razza;
    public int anni;
/** Metodi dinamici public: se senza argomenti, si scrivono
```

```
                    public tipo metodo(){… …}
*/

 public void leggiInput() //metodo dinamico
 {Scanner tastiera = new Scanner(System.in);
/** Definisco un nuovo oggetto "tastiera" di tipo Scanner, capace di
tradurre un input in caratteri inviato da tastiera nella sua
rappresentazione Java. Il metodo nextLine(); se inviato a "tastiera"
richiede una riga di input da tastiera e restituisce la
rappresentazione Java di una stringa. */
  System.out.println( " nome = " );
  nome = tastiera.nextLine(); //nome cane che riceve il metodo
  System.out.println( " razza = " );
  razza = tastiera.nextLine(); //razza cane che riceve il metodo
  System.out.println( " anni = " );
  anni = tastiera.nextInt(); //eta'  cane che riceve il metodo
 }

  public void scriviOutput()//metodo dinamico
  {System.out.println( " nome = " + nome);
   System.out.println( " razza = " + razza);
   System.out.println( " anni = " + anni);}


   public int GetEtaInAnniUmani()//metodo dinamico
//Trasforma gli anni di un cane in corrispondenti per l'uomo.
//Conto 11 anni ciascuno i primi due anni, e 5 anni ogni altro anno
     {if (anni <=2)
        return anni*11;
       else
        return 22 + (anni-2)*5;}
     }

/** Il primo esempio di un programma che usa classi definite da noi.
CaneDemo ha un main, quindi e' un programma, e usa la classe Cane. La
classe Cane deve: (1) trovarsi nello stesso file del programma e non
essere pubblica (e' la nostra scelta), oppure (2) trovarsi in un file
di nome Cane.java della stessa cartella ed essere pubblica.
Per assegnare un attributo pubblico del cane x, per esempio assegnare
"anni" ad 8 devo scrivere: x.anni = 8 */

public class CaneDemo
```

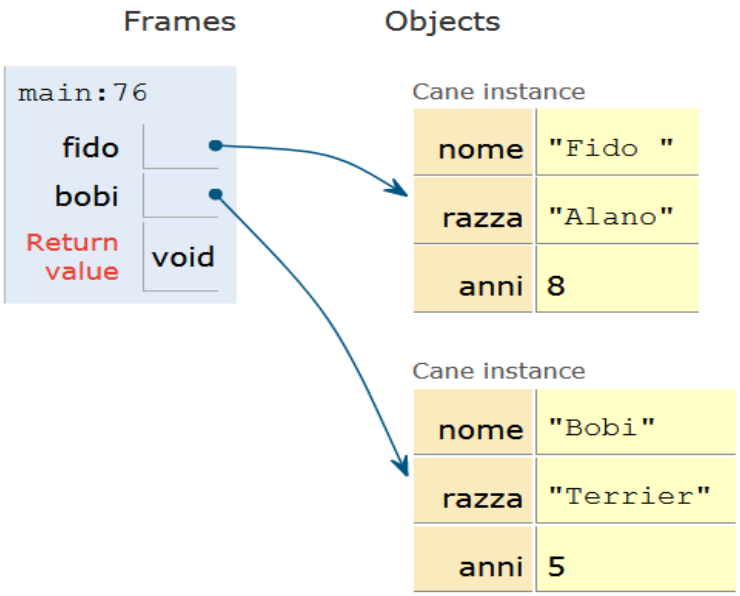
```
/** abbiamo scritto public class CaneDemo, quindi la classe CaneDemo
puo' essere usata da classi in altri files, e deve essere salvata nel
file CaneDemo.java */
{ //Una classe è eseguibile se ha un main, come questo:
  public static void main(String[] args)
  {Cane fido = new Cane();
/** Il comando C x = new C(); definisce un nuovo oggetto x di tipo C
con valori di default per gli attributi. Nel caso di un cane: null,
null, 0 per gli attributi: nome, razza, anni */

    System.out.println( "fido prima inserimento dati" );
    fido.scriviOutput(); // stampo i valori di default: null,null,0
    System.out.println( "Inserisci dati fido" );
    fido.leggiInput();
    System.out.println( "Dati inseriti fido" );
    fido.scriviOutput();// stampo i valori veri
    System.out.println( "eta' fido in anni umani  "
                        + fido.GetEtaInAnniUmani());

    Cane bobi = new Cane(); /** Questo definisce un nuovo cane con
valori di default null, null, 0 per gli attributi nome, razza, anni*/
    System.out.println( "Inserisco dati bobi dentro il programma" );
    bobi.nome = "Bobi";
    bobi.razza = "Terrier";
    bobi.anni = 5;
    bobi.scriviOutput(); }}
/** Fornendo come input:
Fido
Alano
8
Otteniamo come output:
fido prima inserimento dati
 nome = null
 razza = null
 anni = 0
Inserisci dati fido
 nome = Fido
 razza = Alano
 anni = 8
Dati inseriti fido
 nome = Fido
 razza = Alano
```

```
 anni = 8
eta' fido in anni umani  52
Inserisco i dati bobi
 nome = Bobi
 razza = Terrier
 anni = 5 */
```

Qui sotto la situazione della memoria a fine programma. Potete ottenerla con un visualizzatore, per es.:

https://cscircles.cemc.uwaterloo.ca/java_visualize/

Questo visualizzatore vi richiede il programma in un unico file, dunque dovete definire Cane come classe non pubblica.



Nell'immagine vi è rappresentata la struttura del programma. Vi è il main e 2 oggetti: Fido e Bobi appartenenti alla classe cane

Lezione 02 Attributi e metodi privati, get e set

Lezione 2. Parte 1. Un primo esempio di attributi e metodi privati: la classe Specie *(50 minuti)*

Spesso è conveniente o necessario fornire dei metodi che consentano di accedere ai campi di una classe. Tali metodi prendono il nome di ***metodi get*** (per i metodi che leggono il campo di un oggetto) e ***metodi set*** (per i metodi che scrivono il campo di un oggetto). In questo caso gli attributi vengono resi privati: un attributo privato può essere modificato solo dall'interno della classe. Il vantaggio di usare metodi get/set rispetto a rendere pubblici gli attributi della classe è controllare in modo fine le possibilità di accesso (es., omettendo il set si impone il fatto che un attributo sia disponibile solo in lettura) e di impedire modifiche che producono dati contradditori. Di solito gli attributi di una classe sono privati.

Come primo esempio definiamo una classe ***Specie*** con metodi get e set (la trovate sul Savich al cap. 8). Vedremo anche cosa succede assegnando un oggetto di Specie a un altro, e la differenza tra due oggetti identici (che occupano la stessa memoria) e due oggetti uguali attributo per attributo.

```
import java.util.Scanner;
//Specie e' una classe non pubblica e non eseguibile
//per rappresentare delle specie di esseri viventi.
//Scriveremo un programma per sperimentare Specie in una classe
//di nome SpecieDemo, e salveremo tutto nel file: SpecieDemo.java

class Specie /** Classe non pubblica, deve stare nello stesso file
della classe che la usa */
{
/** Rendendo privati gli attributi di Specie, un metodo esterno alla
classe non  puo' piu' modificare direttamente gli attributi:
nome, popolazione, tassoCrescita */
    private String nome;
    private int popolazione;
    private double tassoCrescita;

/** per modificare gli attributi della classe ora e' necessario un
metodo "set": cosi' posso inserire un test per controllare che la
```

```
modifica sia sensata. */
 public void setSpecie(String n, int p, double t)
 {nome = n;
  if (p<0)
  System.out.println( "Valori negativi popolazione non accettati" );
      else popolazione = p;
  tassoCrescita = t;
    }

  /** per ottenere gli attributi della classe ora e' necessario un
metodo "get" */
    public String getNome()           {return nome;}
    public int    getPopolazione()    {return popolazione;}
    public double getTassoCrescita() {return tassoCrescita;}

    public void leggiInput()
    {Scanner tastiera = new Scanner(System.in);

      System.out.println( " nome = " );
      nome = tastiera.nextLine();

      System.out.println( " popolazione = " );
      popolazione = tastiera.nextInt();

      System.out.println( " tasso di crescita = " );
      tassoCrescita = tastiera.nextDouble();
     }

    public void scriviOutput()
    {System.out.println( " nome = " + nome);
     System.out.println( " popolazione = " + popolazione);
      System.out.println( " tasso crescita = " + tassoCrescita);
      }

     public int predicipopolazione(int anni)
     {
       double p = popolazione;
        while(anni > 0)
        {p=p+p*tassoCrescita/100; --anni; }
       return (int) p; //(int) p trasforma il reale p in un intero
      } }
```

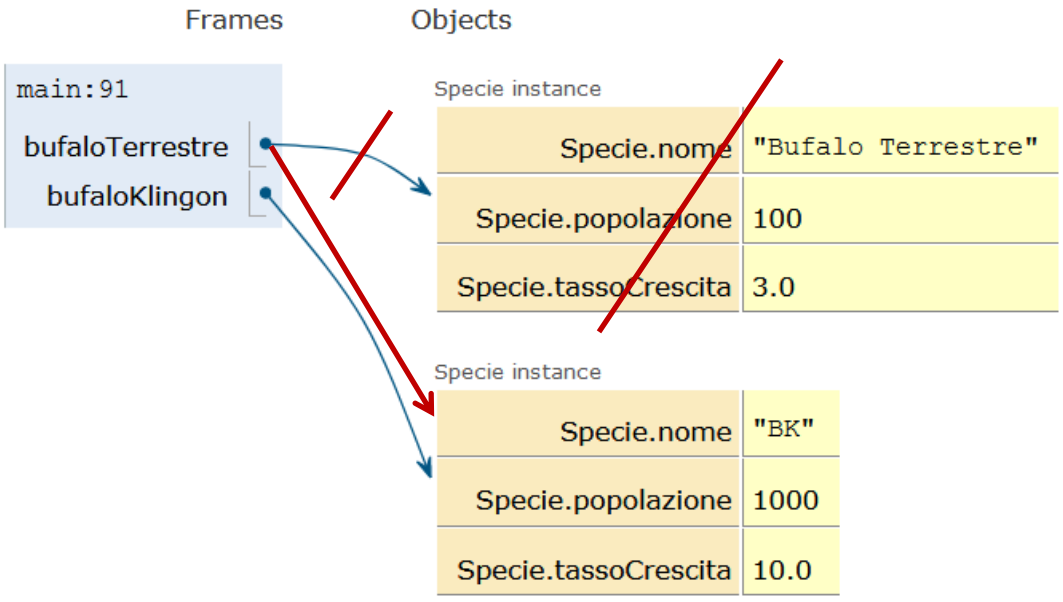
```
/** Introduciamo una classe eseguibile SpecieDemo per sperimentare la
classe Specie. Proviamo a inserire i dati di una specie sia da
tastiera che usando un metodo set. Sperimentiamo cosa succede se
assegnamo un oggetto a un altro. */
public class SpecieDemo
//Classe eseguibile e pubblica, deve stare in: SpecieDemo.java
{
  public static void main(String[] args)
  {Specie bufaloTerrestre = new Specie();

    System.out.println( "BufaloTerrestre prima inserimento dati" );
    bufaloTerrestre.scriviOutput();

    System.out.println( "Inserisci dati BufaloTerrestre" );
    bufaloTerrestre.leggiInput();

    System.out.println( "Dati inseriti BufaloTerrestre" );
    bufaloTerrestre.scriviOutput();

    System.out.println( "BufaloTerrestre dopo 10 anni = "
              + bufaloTerrestre.predicipopolazione(10));

    Specie bufaloKlingon = new Specie();
    System.out.println("Inserisco dati BufaloKlingon usando set");

    /** Non possiamo assegnare nome, popolazione e tasso di crescita
    direttamente perche' questi attributi sono privati */
    bufaloKlingon.setSpecie("BK",1000,10);

    System.out.println( "Dati inseriti BufaloKlingon" );
    bufaloKlingon.scriviOutput();
    System.out.println( "Bufalo Klingon dopo 10 anni = "
                     + bufaloKlingon.predicipopolazione(10));

     System.out.println( "Identifico Bufalo terrestre e Klingon" );
     bufaloTerrestre  = bufaloKlingon;
     bufaloTerrestre.scriviOutput();
     bufaloKlingon.scriviOutput();
  }
}
```

```
/** Effetto dell'assegnazione
                  bufaloTerrestre  = bufaloKlingon;

1. L'indirizzo di bufaloTerrestre diventa uguale a quello di
bufaloKlingon. In altre parole, bufaloTerrestre adesso e' un alias
per bufaloKlingon. Avete visto lo stesso fenomeno quando si assegna
un array a un altro (ProgIB sezione 6.4)

2. abbiamo l'impressione che i dati del bufalo terrestre siano
scomparsi. In realta' sono diventati irraggiungibili: non ho piu' il
loro indirizzo. Java dopo un poco ricicla le aree di memoria
irraggiungibili, che a questo punto spariscono davvero. */
```

Qui sotto la situazione della memoria a fine programma. Potete ottenerla con un visualizzatore, per es.:

https://cscircles.cemc.uwaterloo.ca/java_visualize/

Questo visualizzatore vi richiede il programma in un unico file, dunque dovete definire Specie come classe non pubblica.



Nell'immagine vi è rappresentata la struttura del programma. Vi è il main e 2 oggetti con la relativa classe di appartenenza

Vi invitiamo a fare il seguente esercizio, che spiega perché è bene che gli attributi di una classe siano privati. Definite una classe **Rettangolo** con attributi: ***base, altezza e area***. Un rettangolo è correttamente definito se l'area è uguale a base per altezza. Accedendo dall'esterno posso ignorare che esiste un attributo area e dimenticarmi di modificarlo di conseguenza, creando un rettangolo con dati errati. Provate a definire la classe Rettangolo rendendo tutti gli attributi privati, e scegliete i metodi get e set in modo da impedire una modifica errata dell'area. Trovate la soluzione nel capitolo 8 del Savich, e in queste dispense nella Lezione 03.

Lezione 2. Parte 2. Un esempio non banale di attributi e metodi privati: la classe Calcolatrice. *(50 minuti)*

Vediamo ora un esempio non banale di attributi privati: la classe Calcolatrice. Gli elementi della classe sono calcolatrici tutte uguali, semplici robot virtuali che prendono ricevono una lista di comandi da una tastiera e eseguono dei calcoli.

Una calcolatrice semplificata. consideriamo solo operazioni tra interi (quindi niente tasto "virgola") e argomenti di positivi e di una cifra (quindi non dobbiamo preoccuparci di raccogliere le cifre per formare numeri più grandi). Abbiamo quindi le cifre '0' … '9'. Come uniche operazioni consideriamo + e *. Ogni operazione binaria prende due numeri a e b, insieriti direttamente in memoria oppure risultati degli ultimi due calcoli, li cancella e li rimpiazza con un risultato: a+b oppure a*b.

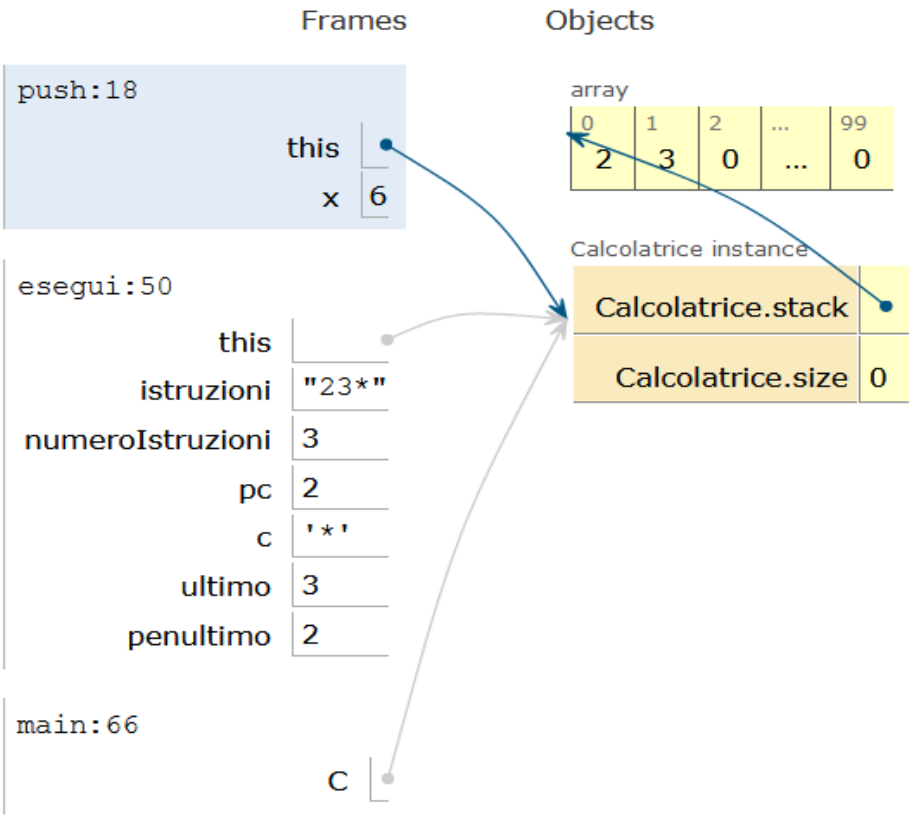
La calcolatrice come oggetto. Di conseguenza una calcolatrice deve avere un vettore ***stack*** di interi per contenere i risultati dei calcoli precedenti non ancora cancellati, e un indice ***size*** per indicare quanti risultati ci sono in stack. Abbiamo bisogno di un metodo dinamico ***int pop()*** che restituisca l'ultimo risultato inserito nella calcolatrice, cancellandolo dal vettore stack, aggiornando size, e un metodo dinamico ***void push(int x)*** che aggiunga un valore x al vettore stack, sempre aggiornando size. Infine un metodo dinamico ***int esegui(String istruzioni)*** che prenda una stringa che rappresenta una lista di tasti premuti e restituisca l'ultimo risultato risultato ottenuto dalla calcolatrice.

Metodi e attributi privati. L'unico metodo che un utilizzatore della calcolatrice ha bisogno di conoscere è "esegui": tutti gli altri

attributi e metodi sono difficili da usare (vedi qui sotto) e quindi è prudente vietarne l'uso da parte di altre parti del programma che non ne conoscono il funzionamento in dettaglio. Renderemo "esegui" ***pubblico*** e tutti gli altri metodi e attributi ***privati***.

```
//Salviamo il tutto nel file CalcolatriceDemo.java
class Calcolatrice //classe non eseguibile e non pubblica
{ /** una calcolatrice e' una pila che contiene fino a 100 interi.
L'ultimo intero e' il risultato delle operazioni fatte finora e la
prossima operazione agisce sugli ultimi due interi a,b e li rimpiazza
con a+b (per una addizione) oppure a*b (per una moltiplicazione) */

/** stack = pila che contiene fino a 100 interi */
  private int[] stack = new int[100];

/** size = numero interi introdotti: all'inizio 0 */
/** le posizioni occupate hanno indice: 0, 1, …, size-1 */
  private int size = 0;

/** push(x): aggiunge un intero x a stack dopo la parte utilizzata
e aumenta la parte di stack utilizzata di uno. */
  private void push(int x)
   { stack[size] = x;  size++; }

/** pop(): restituisce l'ultima intero utilizzato di stack
e lo "cancella", riducendo la parte di stack utilizzata di uno. */
    private int pop()
    {size--; return stack[size];}

/** Un esempio di istruzioni da svolgere: la stringa "23+"
(i)   La prima cifra viene inserita e lo stack passa da: {} (vuoto) a
      {2} (contiene il solo 2).
(ii)  La seconda cifra viene inserita e lo stack passa da {2} a {2,3}
(iii) Quando leggiamo il + togliamo gli ultime due interi dallo stack
      che ritorna ad essere vuoto: {}
(iv)  Sommiamo i due interi: 5=2+3, infine inseriamo 5 nello stack
      che diventa {5} (contiene il solo 5)
(v)   Quando la lista e' finita l'ultimo intero nello stack, 5,
      viene tolto e diventa il risultato */

public int esegui(String istruzioni)
```

```
{int numeroIstruzioni = istruzioni.length(); //lunghezza
 int pc = 0; /** inizio leggendo l'istruzione 0 */
 while (pc < numeroIstruzioni) //eseguo le istruzioni in ordine
  {char c = istruzioni.charAt(pc);  //c = carattere di posto pc
   if (c >= '0' && c <= '9')        //vero se c e' una cifra
    {push(c - '0');} //questa formula mi da' il valore della cifra c
   else if (c == '+')
   {int ultimo    = pop(); //risultato ultimo calcolo
    int penultimo = pop(); //risultato penultimo calcolo
    push(penultimo + ultimo);}
   else if (c == '*' )
   {int ultimo    = pop(); //risultato ultimo calcolo
    int penultimo = pop(); //risultato penultimo calcolo
    push(penultimo * ultimo);}
   pc++; /**  eseguita c passo alla prossima istruzione */
 }
 return pop(); //alla fine restituisco l'ultimo risultato ottenuto
    }
}
```



Nell'immagine vi è la rappresentazione grafica del metodo push relative ad un array
*Lo stato della memoria verso la fine del calcolo di 2*3*

```
//Un esperimento di uso della classe calcolatrice
//Classe eseguibile pubblica, deve stare in CalcolatriceDemo.java
public class CalcolatriceDemo
{public static void main(String[] args)
 {Calcolatrice C = new Calcolatrice();

  System.out.println( "Eseguo istruzioni 23+ (due piu' tre) " );
  System.out.println( C.esegui( "23+" ) + "\n" );

  System.out.println( "Eseguo istruzioni 23* (due  per tre) " );
  System.out.println(C.esegui( "23*" ) + "\n");

  System.out.println(
"Eseguo istruzioni 23*9+ (due  per tre piu' nove) " );
  System.out.println(C.esegui( "23*9+" ) + "\n");

  System.out.println(
"Eseguo istruzioni 99*9* (nove per nove per nove) " );
  System.out.println(C.esegui( "99*9*" ) + "\n");

  System.out.println(
"Eseguo istruzioni 99*9*1+ (nove per nove per nove piu' uno) " );
  System.out.println(C.esegui( "99*9*1+" ) + "\n");}}
```

Lezione 03 Assegnazioni di oggetti, metodo equals,costruttori

Lezione 03. Parte 1 *(30 minuti).* Abbiamo già visto come ogni classe C abbia un costruttore C(): scrivendo new C() costruiamo un nuovo oggetto nella classe C, con tutti gli attributi con valori di default. Questo è il costruttore che abbiamo di default per C. In alternativa, possiamo definire noi stessi dei costruttori pubblici (cap.9.1 del Savich), con le istruzioni

public C(argomenti){ … istruzioni … }

Per un costruttore non indichiamo il tipo di ritorno: già sappiamo che l'oggetto costruito sta nella classe C. Nelle istruzioni assegnamo dei valori agli attributi della classe, utilizzando gli argomenti del metodo: sono i valori del nuovo oggetto che stiamo costruendo. Se definiamo dei costruttori noi stessi il costruttore di default scompare.

Uso di costruttori con lo stesso nome. Tutti i costruttori di una classe hanno il nome della classe e quindi hanno lo stesso nome. In base alla convenzione generale sui nomi, è quindi necessario che due costruttori abbiano un numero di parametri diverso, oppure due parametri di tipo diverso nella stessa posizione. Altrimenti si crea una ambiguità e il costruttore viene rifiutato.

Uguaglianza di oggetti. Per decidere se due oggetti sono uguali dobbiamo definire noi stessi un metodo ***equals*** (cap.8.3 del Savich). Se mandiamo a un oggetto x un metodo equals(y) con parametro un oggetto y otteniamo come risposta true o false, a seconda se x è uguale a y oppure no. Sta a noi decidere quando consideriamo due oggetti di una classe C uguali: in genere controlliamo che tutti gli attributi siano uguali, ma non è l'unica soluzione. È meglio non utilizzare il test x==y di eguaglianza: controlla se x,y hanno lo stesso indirizzo, quindi se occupano la stessa area di memoria. Questo è più di essere uguali, significa che x,y sono due nomi per lo stesso oggetto. In questo caso, diciamo che x,y sono due ***aliases*** (nomi alternativi) per un oggetto.

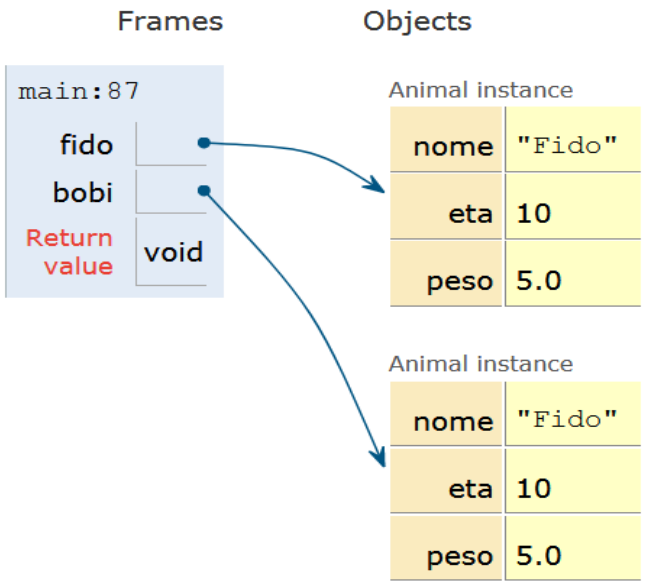
Avete visto il problema di stabilire se due vettori sono uguali nelle dispense di **ProgIB** (sezioni 6.3 e 6.4): il problema di stabilire se due oggetti sono uguali è simile, eccetto che ora usiamo il metodo dinamico equals.

Come esempio, vediamo una classe ***Animal*** per rappresentare gli animali: forniamo ***(i)*** due costruttori, uno con valori di default e uno con valori significativi, ***(ii)*** i metodi get e set, ***(iii)*** un metodo assegna che assegna a un animale gli attributi di un altro, ***(iv)*** un metodo equals per l'eguaglianza attributo per attributo. Come esperimento, definiamo due oggetti di tipo "animale" con attributi uguali e indirizzo diverso.

```
//Inseriamo tutto nel file AnimalDemo.class
class Animal //classe non eseguibile e non pubblica
{ /** Introduciamo una classe per sperimentare costruttori e metodo
equals. Gli attributi sono privati. */
  private String nome;
  private int eta;
  private double peso;

  /** (i) Il primo costruttore assegna valori di default privi di
interesse */
  public Animal(){nome = "nessun nome"; eta=0; peso=0;}

  /** Il secondo costruttore produce un oggetto a partire da
informazioni rilevanti */
  public Animal(String n, int e, double p)
   {nome=n; eta=e; peso=p;}

  /** (ii) Metodi set e get */
  public void setAnimal(String n, int e, double p)
   {nome = n; eta=e; peso=p;}

  public String getNome(){return nome;}
  public int    getEta() {return eta;}
  public double getPeso(){return peso;}

  public void setNome(String n){nome = n;}

  public void setEta(int e)
  {if (e>=0) eta = e;
  else System.out.println("L'eta' deve essere non negativa");}

  public void setPeso(double p)
  {if (peso>=0) peso=p;
    else System.out.println("Il peso deve essere non negativo");}
```

```
  /** Metodo di scrittura */
  public void scriviOutput()
  {System.out.println( " nome " + nome);
   System.out.println( " eta'   " + eta);
   System.out.println( " peso " + peso);}

/** (iii) Metodo che assegna a un animale x gli attributi di un altro
animale y.*/
  public void assegna(Animal altroAnimale)
   {this.nome = altroAnimale.nome;
    this.eta = altroAnimale.eta;
    this.peso = altroAnimale.peso; }
/** Questo metodo di assegnazione e' diverso da assegnare x = y;
nel secondo caso x e y occupano lo stesso spazio di memoria, sono lo
stesso oggetto e ogni modifica fatta a x si ripercuote a y. */

/** (iv) Metodo equals che controlla se due animali hanno gli stessi
attributi. Uso il metodo dinamico s.equalsIgnoreCase(s'): controlla
se s, s' sono uguali ignorando la differenza maiuscole/minuscole.
Qui s,s' sono i nomi di "this" e di "altroAnimale". */
  public boolean equals(Animal altroAnimale)
  {return
    (this.nome.equalsIgnoreCase(altroAnimale.nome))
    &&
    (this.eta  == altroAnimale.eta)
    &&
    (this.peso == altroAnimale.peso);
  }
}

/* Verifichiamo che essere uguali e' diverso dall'avere lo stesso
indirizzo. Usiamo la classe AnimalDemo e il file AnimalDemo.java */
public class AnimalDemo //classe eseguibile pubblica
{
public static void main(String[] args)
{
 Animal fido = new Animal("Fido",10,5.0); /** valori significativi */
 Animal bobi = new Animal(); /** valori di default */

 System.out.println( "1. Fido" );
 fido.scriviOutput();
```

```
 System.out.println( "2. Bobi" );
 bobi.scriviOutput();
 /** All'inizio i due oggetti sono diversi */
 System.out.println("3. Fido e' uguale a Bobi? "+fido.equals(bobi));
 /** Se assegno il primo al secondo attributo per attributo
 diventano uguali attributo per attributo. */
 System.out.println("4. Assegno gli attributi di Fido a Bobi ");
 bobi.assegna(fido);
 System.out.println("5. Fido e' uguale a Bobi? "+fido.equals(bobi));
//Vero: stessi attributi
 System.out.println("6. Fido == Bobi? "+(fido==bobi));
//Falso: diversi indirizzi
}}

/** Qui sotto vedete la situazione alla fine del main: dopo aver
assegnato gli attributi di "fido" a "bobi", i due oggetti hanno gli
stessi attributi ma continuano a occupare aree di memoria differenti
*/
```



Nell'immagine vi è la reppresentazione dell'ereditarietà relativa alla classe animale

Lezione 03. Parte 2 *(30 minuti)*. Vediamo ora, con tre oggetti x,y,z, un esempio della differenza tra assegnare x=y e assegnare ogni attributo di x a z. Nel primo caso x e y diventano due nomi per lo stesso oggetto, ogni cambio fatto ad x si ripercuote su y e viceversa. Nel secondo caso x, z sono indicati come uguali da equals, ma si trovano

in aree di memoria diverse, e se modifico z non modifico x e viceversa. Come esempio ricopiamo e eseguiamo il programma qui sotto, senza spiegare in dettaglio come scriverlo. Si tratta di una variante della classe Specie, a cui aggiungiamo il metodo "cambia" per assegnare a un oggetto gli attributi di un altro oggetto. Nell'esempio, x,y,z sono:

specieTerrestre, specieKlingon, specieAfricana

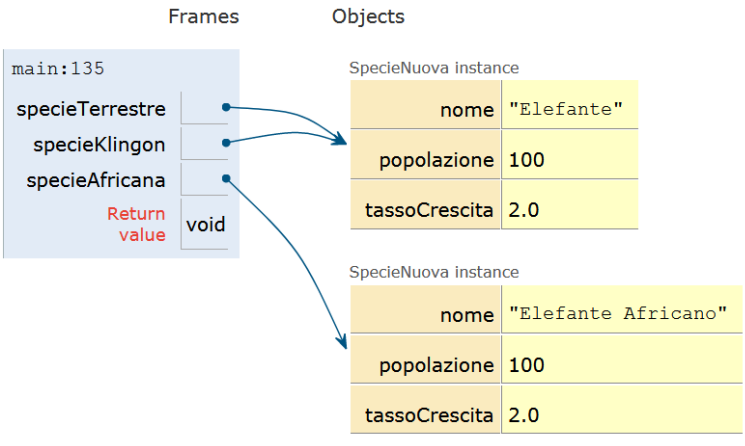
```
//salviamo tutto nel file SpecieNuovaDemo.java
import java.util.Scanner;

class SpecieNuova //classe non eseguibile e non pubblica
{ /** Rendendo privati gli attributi un metodo esterno alla classe
non puo' piu' modificare nome, popolazione, tassoCrescita  */
    private String nome;
    private int popolazione;
    private double tassoCrescita;

  /** per modificare gli attributi della classe ora e' necessario
un metodo "set", che impedisce modifiche errate */
    public void setSpecie(String n, int p, double t)
    {nome = n;
      if (p<0)
        System.out.println( "Valori negativi pop. non accettati" );
      else
        popolazione = p;
      tassoCrescita = t;}

/** per ottenere gli attributi della classe ora e' necessario un
metodo "get" */
    public String getNome()           {return nome;}
    public int    getPopolazione()    {return popolazione;}
    public double getTassoCrescita() {return tassoCrescita;}

    /** I metodi di lettura e scrittura non cambiano */
    public void leggiInput()
    {Scanner tastiera = new Scanner(System.in);

      System.out.println( "   nome = " );
      nome = tastiera.nextLine();

      System.out.println( "   popolazione = " );
```

```
      popolazione = tastiera.nextInt();

      System.out.println( "   tasso di crescita = " );
      tassoCrescita = tastiera.nextDouble();}

    public void scriviOutput()
    { System.out.println( "   nome = " + nome);
      System.out.println( "   popolazione = " + popolazione);
      System.out.println( "   tasso crescita = " + tassoCrescita);}

     public int predicipopolazione(int anni)
     {double p = popolazione;
        while(anni > 0)
        {p=p+p*tassoCrescita/100; --anni; }
       return (int) p;}

/** Per cambiare un oggetto consigliamo di assegnare i suoi
attributi. Questo metodo cambia gli attributi di "altraSpecie".  */
    public void cambia(SpecieNuova altraSpecie)
    {altraSpecie.nome = this.nome;
     altraSpecie.popolazione = this.popolazione;
     altraSpecie.tassoCrescita = this.tassoCrescita;}

/** dobbiamo aggiungere un metodo per confrontare due oggetti: usare
== non va bene, perche' == confronta gli indirizzi dei due oggetti */
     public boolean equals(SpecieNuova altraSpecie)
      {return (nome.equalsIgnoreCase(altraSpecie.nome))
              && (popolazione == altraSpecie.popolazione)
              && (tassoCrescita == altraSpecie.tassoCrescita);}}

//Usiamo una classe SpecieNuovaDemo per sperimentare la classe Specie
public class SpecieNuovaDemo //classe eseguibile pubblica
{private static void pause(){ /** questo metodo aspetta un a capo per
continuare. E' statico, quindi non e' legato a un oggetto. */
 Scanner tastiera = new Scanner(System.in);
 System.out.println( "............. premi a capo per continuare" );
 tastiera.nextLine();}

public static void main(String[] args){
 SpecieNuova specieTerrestre = new SpecieNuova();//primo oggetto
 System.out.println("\n 1. Inserisco specieTerrestre usando un set");
```

```
/** Non possiamo assegnare nome, popolazione e tasso di crescita
direttamente perche' questi attributi sono privati */
 specieTerrestre.setSpecie( "Bufalo Nero" ,500,3);

 System.out.println( "\n 2. Dati inseriti specieTerrestre" );
 specieTerrestre.scriviOutput();
 pause();

 SpecieNuova specieKlingon = new SpecieNuova();//secondo oggetto
 System.out.println("\n 3. Inserisco specieKlingon usando un set");
/** Non possiamo assegnare nome, popolazione e tasso di crescita
direttamente perche' questi attributi sono privati */
 specieKlingon.setSpecie( "Bufalo Klingon" ,1000,10);

 System.out.println("\n 4. Dati inseriti specieKlingon");
 specieKlingon.scriviOutput();
 pause();

 System.out.println("\n 5. Assegno specieterrestre = specieKlingon");
 specieTerrestre  = specieKlingon;
 System.out.println("Ho identificato i due oggetti, ora valgono:");
 specieTerrestre.scriviOutput();
 specieKlingon.scriviOutput();

 System.out.println("\n 6. Per rendermi conto che i due oggetti sono
identificati: " );
 System.out.println("se modifico la specie terrestre in Elefante
modifico anche il  Klingon");
 specieTerrestre.setSpecie("Elefante",100,2);
 specieTerrestre.scriviOutput();
 specieKlingon.scriviOutput();
 pause();

 System.out.println("\n 7. Vediamo ora il modo corretto di modificare
gli oggetti");
 System.out.println("Creo \"specieAfricana\" e le assegno i valori
Elefante");
 SpecieNuova specieAfricana = new SpecieNuova(); //terzo oggetto
 /** invio i dati da specieTerrestre a specieAfricana */
 specieTerrestre.cambia(specieAfricana);
 specieAfricana.scriviOutput();
 pause();
```

```
 System.out.println("\n 8. I primi due oggetti sono lo stesso:
(specieTerrestre == specieKlingon) vale : "
                          + (specieTerrestre == specieKlingon) );
 /** vero, sono lo stesso indirizzo */

 System.out.println("\n 9. Invece il primo e il terzo oggetto no:
(specieTerrestre == specieAfricana) vale : "
                          + (specieTerrestre == specieAfricana) );
 /** falso, hanno gli stessi valori ma non lo stesso indirizzo  */

 System.out.println( "\n 10. Pero' il primo e il terzo oggetto hanno
gli stessi attributi: (specieTerrestre.equals(specieAfricana)) vale :
" + (specieTerrestre.equals(specieAfricana)));
/** vero, hanno gli stessi valori, anche se indirizzo diverso */
pause();

System.out.println("\n 11. Una controprova: modifico la specie
Africana in Elefante Africano.");
specieAfricana.setSpecie( "Elefante Africano",100,2);
specieAfricana.scriviOutput();

System.out.println("\n 12. NON modifico anche la specie  Klingon
perche'  i due oggetti hanno indirizzi diversi: ");
specieKlingon.scriviOutput();   }}
```

Qui sotto vedete la situazione finale, ottenuta con un visualizer. I primi due oggetti si trovano nella stessa area di memoria, il terzo no, e le modifiche sul terzo non si ripercuotono sui primi due.



Nell'immagine vi è la rappresentazione della struttura del programma con il main, gli oggetti con le loro gelative classi di appartenenza e gerarghie di ereditarietà

Soluzione dell'esercizio proposto nella Lezione 2 *(30 minuti).* ***Questa esercizio è piuttosto semplice, e la soluzione è stata omessa nel corso del 2019; ma se trovate il tempo, dateci un'occhiata.*** La classe Rettangolo ha attributi base, altezza e area: l'area dipende da base e altezza. Accedendo dall'esterno posso ignorare che esiste un attributo area e dimenticarmi di modificarlo di conseguenza. Se invece uso metodi get e set, il metodo set aggiorna l'area quando cambio base e altezza.

```
//inseriamo il tutto nel file: RettangoloDemo.java
import java.util.Scanner;

class Rettangolo //classe non eseguibile non pubblica
 { /** Rendendo privati gli attributi, un metodo esterno alla classe
non puo' piu' modificare base, altezza, area */
  private double base;
  private double altezza;
  private double area;

/** per modificare base, altezza e area ora e' necessario un metodo
"set" (uno per attributo, o uno solo per modificare tutti) */
  public void setDimensioni(double b, double h)
   {base = b; altezza = h; area = b*h;}
//Il metodo set aggiorna l'area e non mi consente di modificarla

/** per ottenere base, altezza e area ora e' necessario un metodo
"get" (uno per attributo) */
  public double getBase(){return base;}
  public double getAltezza(){return altezza;}
  public double getArea(){return area;}

  public void leggiInput()
    {Scanner tastiera = new Scanner(System.in);

      System.out.println( " base = " );
      base = tastiera.nextDouble();
      System.out.println( " altezza = " );
      altezza = tastiera.nextDouble();
      area = base*altezza;
     }
```

```
        public void scriviOutput()
    {System.out.println( " base    = " + base);
     System.out.println( " altezza = " + altezza);
     System.out.println( " area    = " + area); }}

//la classe RettangoloDemo, che da' il nome al file
public class RettangoloDemo
{
  public static void main(String[] args)
  {Rettangolo R = new Rettangolo();

    System.out.println( "Inserisci i valori di R" );
    R.leggiInput();

    System.out.println( "Valori inseriti di R" );
    R.scriviOutput();

    System.out.println( "Modifico R con base=altezza=5" );
/**se cerco di assegnare R.base = 5; R.altezza = 5; ottengo un errore
perche' gli attributi base e altezza sono privati. Devo invece
scrivere: */
    R.setDimensioni(5,5);

    System.out.println( "Valori modificati di R" );
    R.scriviOutput();
  }}
```

Lezione 04 La classe "Stack", chiamate di metodi

Lezione 4. Parte 1. Un primo esempio di libreria: la classe Stack *(50 minuti).*

Vi ricordiamo che uno ***stack (o pila)*** è una struttura dati in in cui gli elementi vengono inseriti/rimossi secondo la politica ***FILO (First-In-Last-Out):*** il primo elemento inserito è l'ultimo a essere rimosso. Uno stack nasce vuoto e ha dimensione variabile, con la possibilità di aggiungere un nuovo valore a fine pila oppure di togliere un valore dalla pila e leggerlo. Nella Lezione 02, abbiamo visto uno stack per rappresentare la memoria interna di una calcolatrice. Ora vediamo una classe Stack che provvede uno libreria di operazioni per uno stack di interi, utilizzabile da altre classi. Nella nostra versione uno stack nasce con una capacità massima a nostra scelta, e non può superarla.

Incapsulamento dei dati. Rendiamo privati gli attributi della classe Stack, impedendo così a chi usa la classe di conoscere i dettagli di come è realizzato lo stack. Chi usa la classe conosce solo le operazioni sullo stack importanti per il proprio programma e nient'altro, così non può venire distratto dai dettagli. Questa tecnica di programmazione viene detta ***incapsulamento dei dati*** o ***information hyding***.

Uso di "assert". Alcune operazioni su uno stack producono errore: quando cerchiamo di togliere l'ultimo elemento di uno stack vuoto, oppure di aggiungere un elemento a uno stack che ha raggiunto la capacità massima. In questo caso interrompiamo il programma e spieghiamo l'errore. Per farlo, aggiungiamo una istruzione

assert test: messaggio_di_errore;

dove *test* è una espressione booleana e *messaggio_di_errore* una stringa. ***"assert"*** interrompe l'esecuzione del programma quando il test è falso e invia come spiegazione dell'errore il messaggio che abbiamo scelto, la riga dove si trova l'asserzione che è fallita, e la catena di chiamate che hanno portato all'asserzione.

Abilitazione delle asserzioni. Se usate le asserzioni, fate attenzione se le asserzioni sono abilitate oppure no. Se usate una riga di comando, con **java NomeClasse** eseguite con le asserzioni disabilitate (meno controlli ma più velocità) mentre con **java -ea NomeClasse** eseguite con le asserzioni abilitate (più controlli ma meno velocità). Nel compilatore che usiamo a lezione, Doctor Java,

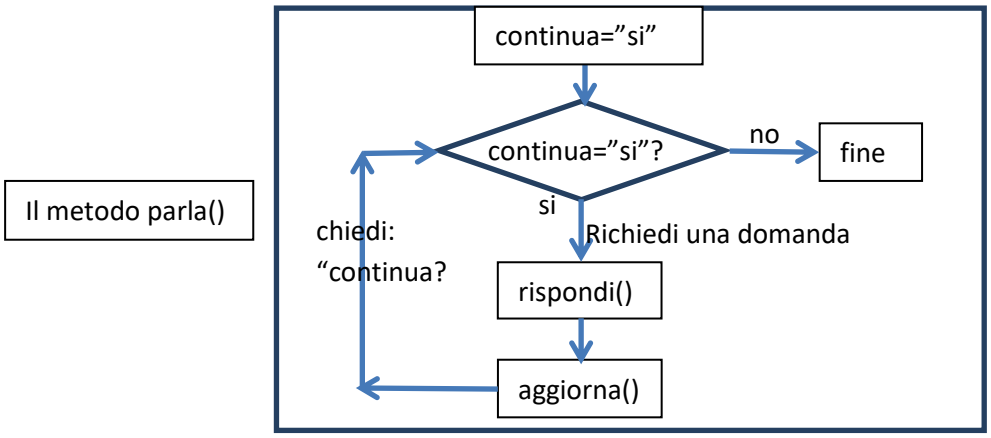
nel menù ***Edit/preferences/miscellaneous*** l'opzione ***"Enable Assert Statement Execution"*** è crocettata di default. Per il vostro Editor, sta a voi controllare come abilitare/disabilitare le asserzioni. Di solito le asserzioni si abilitano nei prototipi, quando cerchiamo gli errori, e si disabilitano nel prodotto finito (tanto a un utente non servono).

```
// Classe per modellare uno stack di interi con capacita' non
// modificabile. Deve essere pubblica, trattandosi di una libreria
// La salviamo in Stack.java

public class Stack
{
 private int[] stack;// inizialmente stack e' il vettore vuoto.
//non fissiamo subito una massima dimensione per tutti gli stack
 private int size=0;
// size=numero elementi inseriti: chiediamo 0 <= size <= stack.length

public Stack(int capacity)
{assert capacity >= 0:
 "la capacita' dello stack doveva essere >=0 invece vale" + capacity;
// adesso fissiamo: massimo numero elementi stack = capacity
 stack = new int[capacity];
// size = numero di elementi inseriti all'inizio e' 0
 size = 0;}

// e' conveniente mettere a disposizione due operazioni per sapere
// se lo stack e' vuoto o pieno. Cio' consente all'utilizzatore
// dello stack di sapere quando un'operazione push/pop e' lecita

public boolean empty(){ return size == 0; }
public boolean full() { return size == stack.length; }

public void push(int x)
{assert !full():
 "tentativo push in uno stack pieno di elementi: " + size;
 stack[size] = x; size++;}

 public int pop()
 {assert !empty():
  "tentativo pop da uno stack vuoto";
  --size; return stack[size];}

/** Per fare esperimenti con gli stacks, definiamo un metodo equals
che controlla se due stack sono identici in tutto: stessa capienza,
stesso numero size di elementi utilizzati, stessi elementi tra quelli
di indice 0, ..., size-1. In questo modo possiamo controllare se uno
```

```
stack contiene gli elementi che dovrebbe contenere */

 public boolean equals(Stack altroStack)
 {if (this.size != altroStack.size) return false;
  if (this.stack.length != altroStack.stack.length) return false;
  int i=0;
  while (i<size)
  {if ((this.stack)[i] != (altroStack.stack)[i]) return false;
   i++;}
  return true;}}



// StackDemo.java  (sperimentiamo la classe Stack)
public class StackDemo
{public static void main(String[] args)
 {
  Stack s = new Stack(3), t = new Stack(3);
  System.out.println("s,t stacks con capacita' 3 entrambi vuoti");
  s.push(10); s.push(20); s.push(30);
  System.out.println("s={10,20,30} pieno, diverso da t={} vuoto");
  System.out.println(" s.full()    = " + s.full());
  System.out.println(" s.empty()   = " + s.empty());
  System.out.println(" s.equals(t) = " + s.equals(t));

  System.out.println("Eliminiamo uno alla volta gli elementi in s");
  System.out.println(" s.pop() = " + s.pop());
  System.out.println(" s.pop() = " + s.pop());
  System.out.println(" s.pop() = " + s.pop());

  System.out.println("Adesso s e' vuoto e uguale a t");
  System.out.println(" s.full()    = " + s.full());
  System.out.println(" s.empty()   = " + s.empty());
  System.out.println(" s.equals(t) = " + s.equals(t));

  System.out.println("Pongo s={40,50} e t={40,60}: s,t diversi");
  s.push(40); s.push(50); t.push(40); t.push(60);
  System.out.println(" s.full()    = " + s.full());
  System.out.println(" s.empty()   = " + s.empty());
  System.out.println(" s.equals(t) = " + s.equals(t));
 }}
```

Lezione 4. Parte 2. Chiamate tra metodi *(40 minuti).* I metodi dinamici possono si richiamare gli uni gli altri, esattamente come succede per i metodi statici che avete visto a ProgI. L'unico vincolo è che queste catene di chiamate devono terminare e il programma deve finire. Ecco un esempio con una classe che contiene ***"oracoli"***, semplici robot virtuali che "dialogano" con l'utente (usando un trucco da prestigiatore).

Un oracolo ha attributi (privati) una "vecchia risposta", una "nuova risposta" una "domanda" (oltre a un oggetto di tipo Scanner per leggere gli input). **I metodi della classe Oracolo si richiamano gli uni gli altri.** Un metodo pubblico *parla()* richiama ciclicamente la richiesta di una domanda, quindi i metodi (privati) *rispondi()* e *aggiorna()*, almeno una volta e finché l'utente vuole continuare.



Nell'immagine vi è rappresentato il flowchart del processo precedentemente descritto

(i) Il metodo rispondi() prima di rispondere chiede un suggerimento all'utente, e lo definisce come "nuova risposta". Per rispondere, invia la "vecchia risposta" all'utente. ***(ii)*** Il metodo aggiorna() assegna la "nuova risposta" alla "vecchia risposta".

```
//salveremo tutto nel file: OracoloDemo.java
import java.util.Scanner;

class Oracolo //classe non eseguibile e non pubblica
{ private String vecchiaRisposta = "la risposta e' nel tuo cuore";
  private String nuovaRisposta;
  private String domanda;
  private Scanner tastiera = new Scanner(System.in);

  public void parla()
  {String continua = "si";
```

```
    while( continua.equalsIgnoreCase( "si" ))
    {System.out.println( "Cosa vuoi chiedere?" );
     domanda = tastiera.nextLine();
     rispondi();
     aggiorna();
     System.out.println( "Vuoi continuare?" );
     continua = tastiera.nextLine();}
    System.out.println( "L'oracolo ora riposa" );
  }

  private void rispondi()
  {System.out.println
    ("Avrei bisogno di un suggerimento: cosa mi suggerisci?");
   nuovaRisposta = tastiera.nextLine();
   System.out.println( "Hai posto la domanda : " + domanda);
   System.out.println( "Ecco la tua risposta : " + vecchiaRisposta);
  }

  private void aggiorna()
  {vecchiaRisposta = nuovaRisposta;}}

public class OracoloDemo
// Classe eseguibile pubblica. Salvare in: OracoloDemo.java
{public static void main(String[] args)
// Costruiamo un singolo oracolo e iniziamo a parlare con lui
  {Oracolo delfi = new Oracolo();
    delfi.parla();}}


/** Il trucco per ottenere un dialogo apparentemente sensato e' il
seguente. La prima risposta e' gia' scelta. Chi pone la domanda
all'oracolo, invia la risposta alla prossima domanda facendo finta di
suggerire la risposta alla domanda presente.

Fornendo i seguenti inputs:
```

```
Quale e' il significato dell'esistenza
Fai la cosa giusta
si
Cosa devo fare nella vita
Conosci te stesso
si
Cosa devo conoscere?
```

```
Non cercare una risposta a tutto
no
```

Si ottengono i seguenti inputs/outputs:

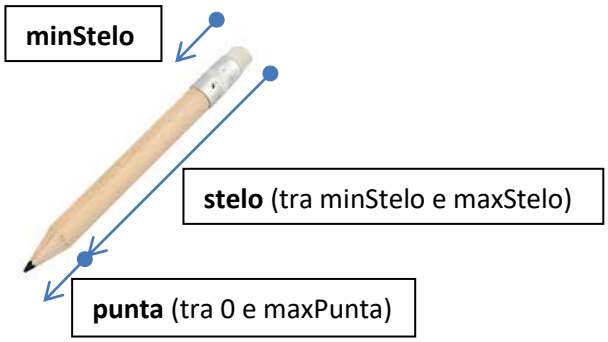
```
Cosa vuoi chiedere?
Qual'e' il significato dell'esistenza
Avrei bisogno di un suggerimento: cosa mi suggerisci?
Fai la cosa giusta
Hai posto la domanda : Qual'e' il significato dell'esistenza
Ecco la tua risposta : la risposta e' nel tuo cuore
Vuoi continuare?
si
Cosa vuoi chiedere?
Cosa devo fare nella vita
Avrei bisogno di un suggerimento: cosa mi suggerisci?
Conosci te stesso
Hai posto la domanda : Cosa devo fare nella vita
Ecco la tua risposta : Fai la cosa giusta
Vuoi continuare?
si
Cosa vuoi chiedere?
Cosa devo conoscere?
Avrei bisogno di un suggerimento: cosa mi suggerisci?
Non cercare una risposta a tutto
Hai posto la domanda : Cosa devo conoscere?
Ecco la tua risposta : Conosci te stesso
Vuoi continuare?
no
L'oracolo ora riposa

*/
```

Esercitazione 01 Attributi e metodi statici e dinamici

In questa esercitazione introduciamo gli attributi statici. Sono indicati da ***public static tipo attributo*** oppure ***private static tipo attributo***. Un attributo statico non descrive un oggetto in particolare, ma l'intera classe. Qui "statico" significa ***"attributo che non fa parte di un oggetto della classe"***, dunque aggiunto alla memoria prima di iniziare l'esecuzione: **non** significa "costante". Invece, per dichiarare un attributo (qualsiasi) costante aggiungete "final": per es. ***public static final tipo attributo***. Per richiamare un attributo/metodo statico pubblico fuori dalla sua classe C scrivete ***C.attributo***, ***C.metodo***. Dentro la classe C scrivete semplicemente ***attributo*** e ***metodo***.

La classe Matita. Vi chiediamo di scrivere una classe pubblica Matita per rappresentare virtualmente matite. Una matita è definita come uno stelo (una lunghezza intera in millimetri, da un minimo ***minStelo*** a un massimo ***maxStelo***) seguita da una punta (un intero da 0 a un massimo ***maxPunta***).

Fissate nella definizione della classe dei valori per massimi e minimi, per esempio: ***minStelo=10, maxStelo=200, maxPunta=5***.



Nell'immagine sovrastante vi è rappresentata una matita divisa in diverse parti: punta, stelo, minStelo

(i) ***minStelo***, ***maxStelo***, ***maxPunta*** sono attributi interi ***pubblici***, ***statici*** e ***final*** della classe Matita (non legati a un oggetto ma alla classe). Invece "stelo" e "punta" sono attributi interi ***dinamici***.

(ii) Il costruttore di ***Matita*** consente di costruire una matita con punta di lunghezza massima dato lo stelo. Un assert impedisce lunghezze non accettabili dello stelo.

(iii) La classe ha i metodi ***get*** per stelo e punta e nessun metodo ***set***: non consento di cambiare la lunghezza a una matita.

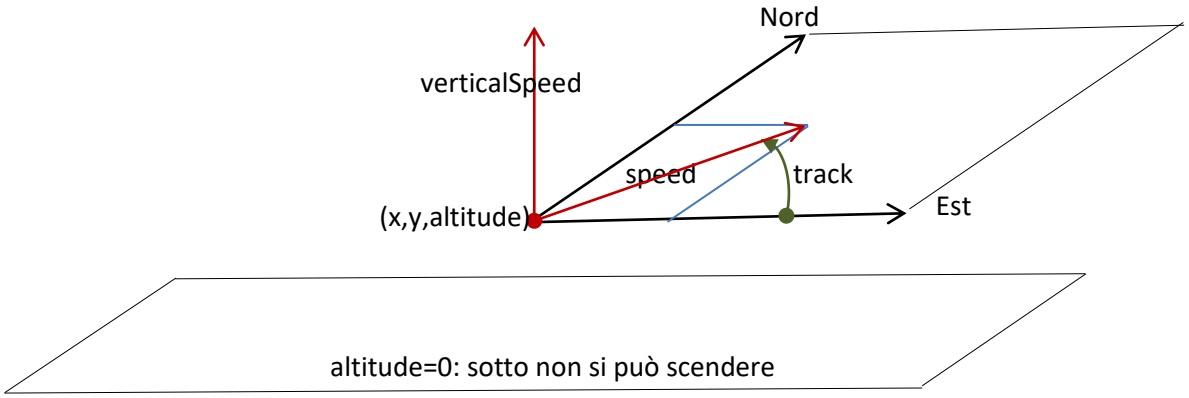
(iv) Un metodo "disegna" restituisce "true" (successo) se la matita ha almeno 1mm di punta, e "false" (fallimento) altrimenti. Nel primo caso usa la matita fino a ridurne la punta di un 1mm.

(v) Un metodo "tempera" restituisce "true" (successo) se la matita è più lunga del minimo e "false" (fallimento) altrimenti. Nel primo caso riduce lo stelo di 1mm e allunga la punta di 1mm, a meno che la lunghezza della punta sia già il massimo. In questo caso la matita si accorcia ma la punta resta invariata.

Scrivete ***Matita.maxStelo*** per richiamare il massimo dello stelo (attributo statico). Scrivete ***Math.min*** per richiamare il metodo statico min(x,y) della classe Math, che calcola il minimo. Includiamo una classe TestMatita per sperimentare la classe Matita: eseguitela e controllate che i risultati siano sensati. Pubblicheremo le soluzioni la prossima settimana.

```
//Una classe come test della classe Matita: salvate nel file
//TestMatita.java        Non si compila senza la classe Matita

public class TestMatita
{public static void main(String[] args)
 {Matita m = new Matita(Matita.maxStelo);
  int s = m.getStelo(), p = m.getPunta();
  System.out.println("Matita di stelo " + s + " e punta " + p);
  System.out.println("Disegno per " + 2*p + " volte:");
  System.out.println("dopo " + p + " volte il disegno fallisce");
  for (int i = 0; i < 2*p; i++)
   System.out.println(" Successo disegno n."+i+" = "+m.disegna());
  System.out.println("Tempero di 1mm la matita"); m.tempera();
  System.out.println(" nuova lunghezza punta = " + m.getPunta());
  System.out.println(" nuova lunghezza stelo = " + m.getStelo());
  System.out.println("Stampo la matita m. Ottengo \"Matita@\" seguito
dall'indirizzo dell'oggetto (in esadecimale): " + m);}}
```

La classe Elicottero. Vi chiediamo di scrivere una classe Elicottero per rappresentare virtualmente elicotteri. Un elicottero è definito con tre coordinate (intere, in km): ***x,y*** e ***altitude*** (non negativa), due velocità (intere, in hm/h), ***speed*** (orizzontale e non negativa) e ***verticalSpeed*** (verticale), e una direzione orizzontale ***track*** (un reale, un angolo in radianti tra 0 e 2π).



Nell'immagine vi è la rapresentazione grafica della posizione e della velocità sugli assi cartesiani

La classe ha i seguenti metodi. Usate degli assert per impedire valori non accettabili degli attributi.

(i) Il costruttore di Elicottero definisce un elicottero fermo in cielo, date le coordinate (x,y,altitude), con velocità nulle e angolo di direzione nullo.
(ii) La classe ha i metodi get per ogni attributo e metodi set per velocità e direzione, ma non per x,y,altitude. Non consentiamo a un elicottero di cambiare la posizione se non spostandosi con lo scorrere del tempo.
(iii) Un metodo ***void elapse(double time)*** modifica la posizione dell'elicottero dato il tempo trascorso, in base alle velocità e alla direzione, usando le formule della trigonometria. Quando assegnate il risultato a delle coordinate intere dovrete arrotondarlo, scrivendo: *(int) espressione*.

Per richiamare un attributo/metodo statico pubblico fuori dalla sua classe C scrivete ***C.attributo***, ***C.metodo***. Per esempio scrivete ***Math.sin***, ***Math.cos*** per i metodi statici per seno e coseno della classe Math. Includiamo una classe ***TestElicottero*** per sperimentare la classe Elicottero: eseguitela (richiede la classe Elicottero) e controllate che i risultati siano sensati. Pubblicheremo le soluzioni la prossima settimana.

```
//Una classe come test della classe Elicottero: salvate nel file
//TestElicottero.java. Non si compila senza la classe Elicottero

public class TestElicottero {

public static void main(String[] args) {
//creo un elicottero collocato all'origine del sistema di riferimento
```

```
 Elicottero a = new Elicottero(0, 0, 0);
// visualizzo le coordinate dell'elicottero
 System.out.println( "Elicottero a di coordinate" );
 System.out.println("("+a.getX()+","+a.getY()+","+
                    a.getAltitude()+")");
// imposto la velocita' dell'elicottero a 500 km/h
 a.setSpeed(500);
// imposto la velocita' verticale dell'elicottero a +10 km/h
 a.setVerticalSpeed(10);
// imposto la direzione dell'elicottero a nord-nord-est (in radianti)
 a.setTrack((double) (3 * Math.PI / 8));
 System.out.println("velocita'="+a.getSpeed()+" velocita' verticale="
                    + a.getVerticalSpeed()+" angolo="+a.getTrack());
// faccio trascorrere mezz'ora
 a.elapse(0.5);
// visualizzo le nuove coordinate: (95,230,5)
 System.out.println("Dopo mezz'ora di volo coordinate: ");
 System.out.println("("+a.getX()+","+a.getY()+","
                     +a.getAltitude()+")");
}}
```

Soluzione dell'Esercitazione 01, es. 1: la classe pubblica Matita.

Come esempio poniamo minStelo, maxStelo, maxPunta uguali a: 5,200,5.

```
// Matita.java
public class Matita {
 public static final int minStelo = 10;  //min. lunghezza matita (mm)
 public static final int maxStelo = 200; //max. lunghezza matita (mm)
 public static final int maxPunta =   5; //max. lunghezza punta  (mm)
//Una matita e' uno stelo seguito da una punta
 private int stelo; // 0 <= stelo <= maxStelo
 private int punta; // 0 <= punta <= maxPunta

 public Matita(int stelo) {
 assert minStelo<=stelo && stelo<=maxStelo:
 "stelo matita non accettabile:" + stelo;
 this.stelo = stelo;
 this.punta = maxPunta;}

/** "disegna" restituisce true quando la matita ha ancora punta, e ne
riduce la punta di 1 mm. Restituisce false se la punta e' finita. */
    public boolean disegna() {
 if (this.punta > 0)
    {this.punta--;
     return true;}
 else
     return false;}

// "tempera" riduce di un 1mm la matita, e allunga di 1mm la punta
// a meno che la lunghezza della punta sia gia' il massimo
    public boolean tempera() {
 if (this.stelo > minStelo) {
     this.stelo--;
     this.punta = Math.min(this.punta + 1, maxPunta);
     return true;}
  else
     return false;}

 public int getStelo() {return this.stelo;}
 public int getPunta() {return this.punta;}}
```

Soluzione dell'Esercitazione 1, es. 2: la classe pubblica Elicottero.

```
//Elicottero.java
public class Elicottero {
private int x;
private int y;
private int altitude; // 0 <= altitude
private int speed;    // velocita' orizzontale: 0 <= speed
private int verticalSpeed;
private double track;  // angolo direzione: 0<= track<=2pigreco

/** Costruiamo un elicottero sospeso nelle coordinate (0,0,altitude)
con velocita' nulle e angolo di direzione 0 */
public Elicottero(int x, int y, int a) {
 this.x = x;
 this.y = y;
 this.altitude = a;
 this.speed = 0;
 this.verticalSpeed = 0;
 this.track = 0.0;}

//Metodi get per ogni attributo
public int getX()              {return x;}
public int getY()              {return y;}
public int getAltitude()       {return altitude;}
public int getSpeed()          {return speed;}
public int getVerticalSpeed() {return verticalSpeed;}
public double getTrack()       {return track;}

/** Metodi set per speed, verticalSpeed e track. Non consento invece
di cambiare le coordinate "istantaneamente" */
public void setSpeed(int speed) {
 assert 0 <= speed: "velocita' non accettabile:" + speed;
 this.speed = speed;}

public void setVerticalSpeed(int verticalSpeed)
 {this.verticalSpeed = verticalSpeed;}

public void setTrack(double track)
{assert 0 <= track && track <= 2 * Math.PI:
 "angolo non accettabile:" + track;
 this.track = track;}
```

```
/** Consento di cambiare le coordinate con lo scorrere del tempo */
public void elapse(double time) {
 int dx = (int) (speed * Math.cos(track) * time); //incremento x
 int dy = (int) (speed * Math.sin(track) * time); //incremento y
 int dz = (int) (verticalSpeed * time);           //incremento z
 x += dx;
 y += dy;
 altitude = Math.max(0, altitude + dz); }
}
```

Lezione 05 Modelli di oggetti reali e Information Hyding

Lezione 05. Parte 1. Metodi statici e dinamici *(50 minuti).* Alcune classi si definiscono più facilmente con metodi statici, non legati a oggetti. Per introdurre un attributo/metodo statico pubblico in una classe C scriviamo ***public static tipo attributo*** e ***public static tipo metodo(…){…}***. Cambiando ***public*** in ***private*** rendiamo l'attributo/metodo privato nella classe C. Dato che un attributo/metodo statico non è legato a un oggetto, per richiamarlo nella sua classe C scriviamo semplicemente ***attributo, metodo***. Per richiamarlo fuori dalla sua classe C (in questo caso deve essere pubblico) scriviamo invece ***C.attributo, C.metodo***, per indicare in quale classe C cercarlo. Un esempio: il metodo statico min(x,y) per il minimo si trova nella classe Math, e lo richiamiamo scrivendo ***Math.min***.

Come esempio, definiamo una classe ***Bottiglia*** con metodi dinamici per rappresentare l'oggetto fisico bottiglia e le operazioni su di esso. Presentiamo un indovinello (***Die Hard Water Jug Riddle***, qui sotto) e le operazioni su bottiglie consentite per risolverlo con una classe ***DieHard*** definita a partire dalla classe Bottiglia. In DieHard non introduciamo oggetti, dunque definiamo ***solo metodi statici*** su bottiglie. Terminiamo risolvendo l'indovinello usando i metodi della classe DieHard. Ecco l'indovinello del film Die Hard.

*«**Die Hard 3: the Water Jug Riddle**. Nel film Die Hard 3, i nostri eroi, John McClane (Bruce Willis) e Zeus (Samuel L. Jackson) devono ottenere esattamente quattro galloni da due bottiglie di cinque e tre galloni, per risolvere un enigma dal malvagio Peter Krieg (Jeremy Irons). Possono **riempire** o **svuotare** una bottiglia o **travasare** da una bottiglia in un'altra **finché non svuotano la bottiglia o riempiono l'altra**.»*

Per modellare la soluzione dell'indovinello, definiamo prima una classe ***Bottiglia***. Un oggetto "bottiglia" ha una capacità (non modificabile) e un livello (modificabile). Ci sono i metodi get, il metodo set per il solo livello, e un metodo di stampa. Ci sono metodi ***"aggiungi"*** e ***"rimuovi"*** per aggiungere e rimuovere una quantità a una bottiglia per quanto possibile (fino a quando la bottiglia è piena o vuota): questi metodi restituiscono la quantità effettivamente aggiunta e tolta. Controlliamo con un assert di non scendere sotto zero e di non superare la capacità.

```
//Bottiglia.java
/** Versione con: uso assert, con this omesso ove possibile, con
metodi get e set. Per evitare modifiche alla capacita' non forniamo
un metodo set per la capacita'. */

public class Bottiglia{ //Nota: quantita' intere espresse in galloni
 private int capacita; // 0 <= capacita
 private int livello;  // 0 <= livello <= capacita'

 public Bottiglia(int capacita)
 {this.capacita = capacita;
  livello = 0;
  assert (0<=livello) && (livello <= capacita);}
/** Qui l'uso di this e` fondamentale se vogliamo usare la variabile
capacita sia come argomento del metodo che come attributo */

/** aggiungiamo tutta la parte di una quantita' data che trova posto
nella bottiglia (dunque il minimo tra la quantita' data e la
capacita' residua). Restituiamo la quantita che abbiamo aggiunto
(che puo' essere meno della richiesta)  */
 public int aggiungi(int quantita)
 {assert quantita >= 0:
  "la quantita' doveva essere >=0 invece vale " + quantita;
  int aggiunta = Math.min(quantita, capacita-livello);
  livello = livello + aggiunta;
  assert (0<=livello) && (livello <= capacita);
  return aggiunta;}
/** min e' un metodo statico della classe Math, quindi fuori dalla
classe Math lo indico con Math.min */

/** Rimuoviamo da una bottiglia una quantita' richiesta se c'e',
altrimenti togliamo  tutto (dunque il minimo tra la quantita'
richiesta e il livello). Restituiamo la quantita' rimossa
(che puo' essere meno della  richiesta) */
 public int rimuovi(int quantita)
 { assert quantita >= 0:
  "la quantita' doveva essere >=0 invece vale " + quantita;
  int rimossa = Math.min(quantita, livello);
  livello = livello - rimossa;
  assert (0<=livello) && (livello <= capacita);
  return rimossa;}
```

```
 public int getCapacita(){ return this.capacita; }
 public int getLivello() { return this.livello; }
// Non consentiamo di cambiare la capacita'

 public void setLivello(int livello)
 {this.livello = livello;
  assert (0<=livello) && (livello <= capacita);}

 public void scriviOutput()
 {System.out.println(livello + "/" + capacita);}}
```

Rappresentiamo le operazioni lecite dell'indovinello ***The Water Jug Riddle*** in modo object-oriented con tre metodi statici: ***"riempi, svuota, travasa"*** di argomenti di tipo Bottiglia. La definiziome di "travasa" è delicata: possiamo togliere da una prima bottiglia fino a quanto ci sta in una seconda bottiglia, ma dobbiamo fermarci non appena la prima bottiglia è vuota. Se usiamo i metodi "rimuovi" e "aggiungi" della classe Bottiglia, ci impediamo di togliere da una bottiglia già vuota e di aggiungere a una bottiglia già piena.

```
// DieHard.java
public class DieHard{ /** Non costruiamo oggetti per DieHard.
Dunque i metodi di DieHard non vengono inviati a oggetti della classe
e devono essere dichiarati statici. */

 public static void riempi(Bottiglia b)
 { //riempo fino al massimo livello consentito la bottiglia b
  b.setLivello(b.getCapacita());}

 public static void svuota(Bottiglia b){b.setLivello(0);}

 public static void travasa(Bottiglia a, Bottiglia b)
 { //calcolo quanto basta per riempire b
  int capienzaResiduaB = b.getCapacita() - b.getLivello();
/** rimuovo questa quantita da a, rimuovo tutto da a se a non basta a
riempire b. Aggiungo la quantita' ottenuta alla bottiglia b */
   b.aggiungi(a.rimuovi(capienzaResiduaB));}

 public static void descrizione(String m, Bottiglia b3, Bottiglia b5)
 {System.out.println(m);b3.scriviOutput();b5.scriviOutput();}
```

```
  public static void main(String[] args){
//Una soluzione all'indovinello con le tre operazioni consentite
  Bottiglia b3 = new Bottiglia(3), b5 = new Bottiglia(5);
                 descrizione("Inizio",          b3,b5);
  riempi(b5);    descrizione("Riempio b5",      b3,b5);
  travasa(b5, b3); descrizione("Travaso b5 su b3",b3,b5);
  svuota(b3);    descrizione("Svuoto b3",       b3,b5);
  travasa(b5, b3); descrizione("Travaso b5 su b3",b3,b5);
  riempi(b5);    descrizione("Riempio b5",      b3,b5);
  travasa(b5, b3); descrizione("Travaso b5 su b3",b3,b5);}}
```

The Water Jug Riddle

Lezione 05. Parte 2. Un esempio di incapsulamento di dati: la classe Frazione. *(40 minuti).* Riprendiamo l'argomento di incapsulamento dei dati, e definiamo una classe Frazione per rappresentare le frazioni intere e le operazioni su di esse. Nella classe, una frazione viene sempre ridotta ai minimi termini e con numeratore positivo. Per evitare che chi usa la classe si dimentichi di questa regola rendiamo privati gli attributi numeratore e denominatore. Non inseriamo metodi get, perché possono causare confusione: numeratore e denominatore di una frazione possono essere diversi dai valori che abbiamo assegnato. Inseriamo un metodo set che assegna ***contemporaneamente*** numeratore e denominatore, lasciando alla classe il compito di semplificare la frazione.

Una frazione viene definita come una espressione (num/den) con den>0. Per ridurre una frazione ai minimi termini dividiamo numeratore e denominatore per MCD(num,den): dunque 6/4 diventa 3/2. Cambiamo segno a entrambi se il numeratore è negativo: 1/(-2) diventa (-1)/2. Calcoliamo somma e prodotto con le formule ***(a/b + c/d) = (ad + bc)/bd*** e ***(a/b)(c/d)=(ac)/(bd)***. Due frazioni (a/b) e (c/d) sono uguali se ad=bc.

Altri possibili metodi per la classe Frazione. Se due frazioni sono ridotte ai minimi termini e hanno denominatore>0, per controllare se sono uguali basterebbe controllare se a=c e b=d. Potremmo aggiungere alla classe metodi per: il calcolo del valore reale arrotondato, per la differenza, e per la divisione con divisore diverso da 0.

Proprietà invariante per la classe Frazione. Un ***invariante*** di una classe è una proprietà inizialmente vera per ogni oggetto costruito e che viene preservata da ogni applicazione di ogni metodo. Ogni classe ha molti invarianti. Un invariante significativo di Frazione è: ***ogni frazione è ridotta ai minimi termini con denominatore positivo.***

```
//Frazione.java
public class Frazione
{ /** frazioni ridotte ai minimi termini e con denominatore >0*/
 private int num, den;
/** Metodo statico per il calcolo del Massimo Comun Divisore per
interi a,b non entrambi nulli. */
 private static int MCD(int a, int b)
 {int r; while (b!=0) {r=a%b;a=b;b=r;} return Math.abs(a);}
/** Questo algoritmo viene detto l'algoritmo di Euclide */

/** Il prossimo metodo semplifica la frazione e rende positivo il
denominatore */
 private void semplifica()
 {int m=MCD(num,den); num=num/m;den=den/m;
  if (den<0){num=-num;den=-den;} /** Rendo il denominatore > 0 */}

/** Costruttore: crea una frazione, la semplifica, e forza il
denominatore a essere positivo. */
 public Frazione(int num, int den)
 {assert den!=0: "denominatore frazione deve essere diverso da 0";
/** blocchiamo l'esecuzione quando il numeratore e' 0 */
  this.num=num; this.den=den; this.semplifica();}

/** metodo set: anche questo metodo semplifica e rende positivo il
denominatore */
 public void setFrazione(int n, int d)
 {assert den!=0: "denominatore frazione deve essere diverso da 0";
/** blocchiamo l'esecuzione quando il numeratore e' 0 */
  this.num=num; this.den=den; this.semplifica();}
```

```
/** Metodo di scrittura */
 public void scriviOutput()
 {if (den != 1)
   System.out.println(num + "/" + den);
  else //allora den=1, e al posto di (num/1) scrivo num
   System.out.println(num);}

/** metodo di eguaglianza: funziona anche se la frazione non e'
semplificata */
 public boolean equals(Frazione f)
 {return (this.num * f.den == this.den * f.num);}

/** metodo di somma: il risultato viene creato semplificato */
 public Frazione somma(Frazione f)
 {int n= this.num*f.den + this.den * f.num; int d= this.den*f.den;
   return new Frazione(n,d);}

/** metodo di prodotto: il risultato viene creato semplificato */
 public Frazione prodotto(Frazione f)
 {int n= this.num*f.num; int d= this.den*f.den;
  return new Frazione(n,d);}}


//FrazioneDemo.java   (esperimenti su frazioni)
public class FrazioneDemo{public static void main(String[] args)
{Frazione t = new Frazione(2,3), u = new Frazione(1,6),
 v = new Frazione(1,6); //t=2/3, u=1/6, v=1/6
 System.out.println( "t,u,v valgono" );
 t.scriviOutput();u.scriviOutput();v.scriviOutput();

//t+u+v=(2/3)+(1/6)+(1/6)=((4+1+1)/6)=(6/6)=1
 System.out.println( "t+u+v deve fare 1:" );
 Frazione w = (t.somma(u)).somma(v); w.scriviOutput();

//t*u*v=((2*1*1)/(3*6*6))=(2/108)=(1/54)
 System.out.println( "t*u*v deve fare (1/54)" );
 Frazione z = (t.prodotto(u)).prodotto(v); z.scriviOutput();}}
```

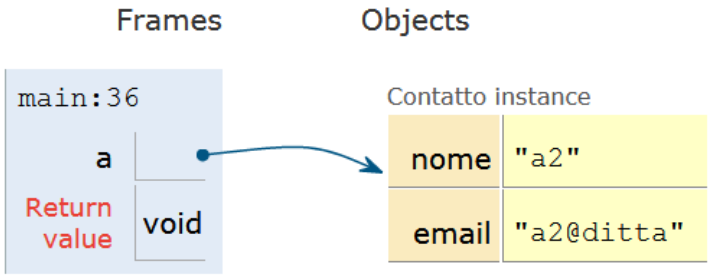
Lezione 06 Classi di vettori di oggetti

Lezione 06 *(90 minuti)*. In questa lezione definiamo Rubrica, il primo esempio di classe di vettori di oggetti (vedi cap.9.6 del Savich). Un contatto è la coppia di un nome e di un indirizzo e-mail e una Rubrica è un vettore di contatti.

La classe Contatto ha attributi privati, metodi get e set e un metodo di scrittura.

```
//Contatto.java
//contiene: costruttore a 2 argomenti, metodi get, set e output
public class Contatto
{//un contatto e' la coppia di un nome e del suo indirizzo email
 private String nome;
 private String email;

 public Contatto(String nome, String email)
 {this.nome = nome; this.email = email;}

 public  String getNome() {return nome;}
 public  String getEmail(){return email;}

 public void setNome(String n){nome = n;}
 public void setEmail(String e){email = e;}

 public void scriviOutput()
 {System.out.println( " - " + nome + " : " + email);}
}


//ContattoDemo.java
public class ContattoDemo
{//controllo i metodi della classe Contatto
  public static void main(String[] args)
  {Contatto a = new Contatto( "a", "a@ditta");
    System.out.println( "Contatto a" );
    a.scriviOutput();
    System.out.println( "Modifico nome a in a2" );
    a.setNome( "a2" );
    a.scriviOutput();
    System.out.println( "Modifico email a in a2@ditta" );
```

```
    a.setEmail( "a2@ditta" );
    a.scriviOutput();
  }
}
```



Il contatto {"a","a@ditta"} dopo che abbiamo modificato nome ed e-mail

La classe Rubrica ha come attributi privati: ***(i)*** un vettore "contatti" di contatti parzialmente riempito, e ***(ii)*** un intero "numeroContatti" che ci dice quanti contatti sono stati inseriti nel vettore.

Rubrica ha un costruttore public Rubrica(int maxContatti){…} per definire una rubrica di massimo numero di contatti pari a maxContatti. Una volta scelto, il massimo di contatti non cambia.

Rubrica ha un metodo int getNumContatti() per ottenere il numero di contatti inseriti. Rubrica non ha un metodo get per il vettore dei contatti né ha metodi set. Questo per evitare che una modifica dei contatti dall'esterno produca delle contraddizioni, per esempio: più contatti di quanti ne indichi il metodo getNumContatti().

Un ***invariante*** di una classe è una proprietà inizialmente vera per ogni oggeto costruito e che viene preservata da ogni applicazione di ogni metodo. Definiamo Rubrica in modo da rendere vero il seguente invariante: ***ogni nome compare in al più un contatto di una rubrica e (0 ≤ numContatti ≤ lunghezza vettore contatti).*** In questo modo ogni nome definisce al più un indirizzo e-mail in ogni rubrica, e non abbiamo il problema di quale indirizzo e-mail scegliere per un dato nome.

Rubrica ha un metodo privato int cercaIndice(String n) per ottenere l'unica posizione di un nome (identifichiamo i nomi a meno di maiuscole/minuscole). Restituisce numContatti se il nome non compare. Il metodo è privato per impedire un accesso a un contatto non controllato da un metodo.

Rubrica ha metodi pubblici:

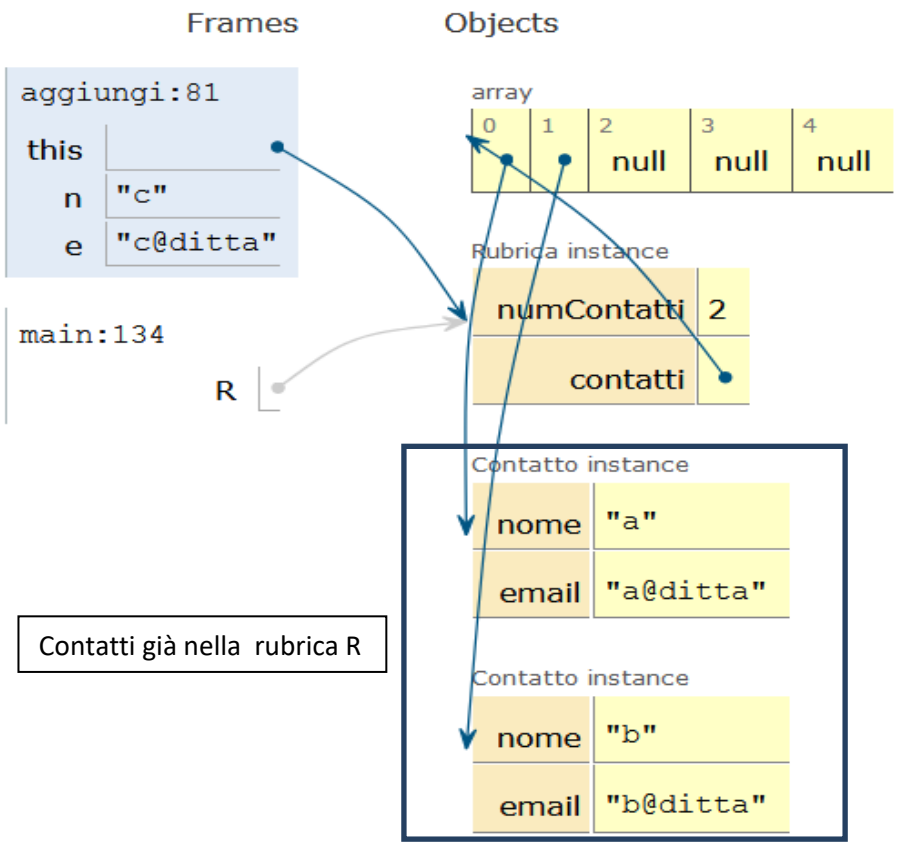
- ***(i)*** ***void scriviOutput()*** stampare la rubrica,
- ***(ii)*** ***boolean presente(String n)*** per decidere se un nome è presente,
- ***(iii)*** ***String cercaEmail(String n)*** per cercare un indirizzo e-mail dato il nome (restituisce "" se il nome non c'è)
- ***(iv)*** ***boolean piena()*** per decidere se la rubrica è piena,
- ***(v)*** ***boolean aggiungi (String n, String e)***
  boolean rimuovi (String n)
  boolean cambiaNome (String n, String n2)
  boolean cambiaEmail(String n, String e2)
  per aggiungere, togliere o modificare contatti da una rubrica.

Per ogni azione del punto ***(v)*** controlliamo prima se l'invariante viene preservato. Se è così l'azione viene eseguita e restituiamo ***true***, se non è così l'azione non viene eseguita e restituiamo ***false***.

```
//Rubrica.java
public  class Rubrica
{ /** Invariante: (i) una volta costruita, una rubrica non contiene
lo stesso nome due volte, (ii) (0<=numContatti <= lunghezza vettore
contatti) */

private int numContatti;          //all'inizio vale 0
private Contatto[] contatti;      //all'inizio vale null

public Rubrica(int  maxContatti){
// costruisce una rubrica con max. num. di contatti  = maxContatti
 numContatti =  0;
//all'inizio i contatti significativi nella rubrica sono 0
 contatti =  new Contatto[maxContatti];
/** all'inizio tutti i contatti nella rubrica non sono significativi:
hanno nome e email uguali a null */
  }
//La nuova rubrica costruita soddisfa l'invariante

public  int getNumContatti(){return numContatti;}
/** non diamo un metodo get per ottenere il vettore dei contatti:
conoscendolo, un'altra classe potrebbe leggere e modificare i
contatti in modo errato (in contraddizione con l'invariante) */

public void scriviOutput()
```

```
 {int i=0;
  System.out.println( "Num. contatti = " + numContatti);
//Stampiamo i contatti di indice da 0 fino a numContatti-1.
//Gli altri contatti sono privi di significato
    while(i<numContatti){contatti[i].scriviOutput();++i;}}

/** Il metodo cercaIndice(n) restituisce l'unico indice i di un
contatto di nome n se c'e', restituisce numContatti se non c'e'. Il
metodo cercaIndice(n) e' privato per evitare che le altre classi
modifichino un contatto in contraddizione con l'invariante */

private int cercaIndice(String n)
{int i=0;
/** Esaminiamo i contatti di indice da 0 a numContatti-1: il primo
con nome n e' il contatto cercato */
while(i < numContatti)
{if (contatti[i].getNome().equalsIgnoreCase(n)) return  i; ++i;}
//Se non troviamo n restituiamo un valore fittizio: numContatti
    return numContatti;}

/** usando cercaIndice(n) possiamo stabilire se il nome n e' presente
nella rubrica */
public   boolean presente(String n)
{return (cercaIndice(n) < numContatti);}

/** usando cercaIndice(n) possiamo trovare quale e-mail corrisponde a
un nome presente nella rubrica (restituiamo "" per nome assente) */
public String cercaEmail(String n)
{int i=cercaIndice(n);
 if (i<numContatti) return contatti[i].getEmail(); else return "";}

/** controlliamo se una rubrica e' piena, cioe' se il numContatti e'
uguale al numero di elementi che possiamo inserire nel vettore
contatti */
  public   boolean piena()
   {return (numContatti  == contatti.length);}

/** Ora possiamo definire metodi per aggiungere, rimuovere e cambiare
contatti. I metodi restituiscono false quando falliscono */
  public   boolean  aggiungi(String n, String e)
   {if (presente(n)) return false; //rubrica contiene n: fallimento
   if (piena()) return false;      //rubrica piena: fallimento
```

```
   contatti[numContatti] = new Contatto(n,e);
//aggiungo il nuovo contatto nella prima posizione disponibile
     ++numContatti; //aggiorno il numero degli elementi
     return true;   //successo
  }

/** Per rimuovere un contatto sposto l'ultimo contatto al posto del
contatto rimosso */
  public boolean  rimuovi(String n)
  {int i=cercaIndice(n);
   if (i==numContatti) return false;//il contatto manca: fallimento

/** se invece il contatto c'e': diminuiamo di 1 il numero dei
contatti e spostiamo il contatto al fondo (ora irraggiungibile) al
posto del contatto i da cancellare. Se i e' l'ultimo contatto allora
i resta al fondo e resta non piu' raggiungibile. */
   --numContatti;
   contatti[i]=contatti[numContatti];
   return true; //successo
  }

//cerco un contatto di nome n e se lo trovo cambio il nome a n2
  public boolean cambiaNome(String n, String n2)
   {int i = cercaIndice(n), j = cercaIndice(n2);
    if ((i == numContatti) || (j<numContatti)) return false;
//contatto di nome n non trovato oppure di nome n2 trovato:fallimento
//Altrimenti cambiamo il nome del contatto i da n a n2
    contatti[i].setNome(n2);
    return true;}

//cerco un contatto di nome n e se lo trovo cambio la email a e2
public boolean cambiaEmail(String n, String e2)
 {  int i = cercaIndice(n);
    if (i == numContatti) return false;
//contatto di nome n non trovato: fallimento
//se n e' presente modifichiamo la email
    contatti[i].setEmail(e2);
    return true;}}
```



Aggiunta del contatto {"c","c@ditta"} alla rubrica R={{"a","a@ditta"},{"b2","b@ditta"}}

```
//RubricaDemo.java
public class RubricaDemo {public static void main(String[] args)
//Consentiamo 3 elementi in rubrica e proviamo a inserirne 4
 {Rubrica R = new Rubrica(3);
  System.out.println("(1) Rubrica con contatti a,b,c:" );
  R.aggiungi( "a", "a@ditta"); R.aggiungi( "b", "b@ditta");
  R.aggiungi( "c", "c@ditta"); R.aggiungi( "d", "d@ditta");
  R.scriviOutput(); //troviamo a,b,c: l'aggiunta di d e' fallita.
  System.out.println( "e-mail di c=" + R.cercaEmail( "c" ));

  System.out.println( "(2) Rimuovo a" );
  R.rimuovi( "a" ); R.scriviOutput();
  System.out.println( "(3) Aggiungo b (ma c'e' gia'): successo = "
  + R.aggiungi( "b", "e")); R.scriviOutput();
  System.out.println( "(4) Modifico b in b2: successo = "
  + R.cambiaNome( "b", "b2")); R.scriviOutput();
  System.out.println( "(5) Modifico b@ditta in b2@ditta: successo = "
  + R.cambiaEmail( "b2", "b2@ditta")); R.scriviOutput();}}
```

Fine Lezione 06. Nota sulla rimozione di un contatto da una rubrica e alias (attenzione, non fa parte del corso). Nelle lezioni precedenti abbiamo parlato del fenomeno della condivisione di memoria: se la variabile x indica un oggetto allora x è l'indirizzo di un'area di memoria e se assegnamo y=x allora x e y sono due indirizzi della stessa area di memoria, quindi indicano lo stesso oggetto. In altre parole: se modifichiamo x modifichiamo anche y. In inglese due indirizzi x e y per lo stesso oggetto nell'area raggiungibile della memoria vengono detti ***alias***. Quando si inizia a programmare, gli alias sono da evitare per quanto possibile perché conducono facilmente a errori. Un alias fa sì che quando modifichiamo un oggetto senza saperlo ne modifichiamo un altro.
Nel 2018, uno studente mi ha chiesto se il metodo ***boolean rimuovi(String n)*** visto in questa lezione e che rimuove un contatto da una rubrica crea indirizzi duplicati per lo stesso contatto o alias. Ricopiamo qui il metodo:

```
public  boolean  rimuovi(String n)
{int i = cercaIndice(n);
 if (i == numContatti) return false;
 --numContatti;
 while (i < numContatti){contatti[i]=contatti[i+1]; ++i;}
 return true;}
```

La risposta è: no, il metodo ***rimuovi non crea degli alias***. Il metodo ***rimuovi*** trova un indice i in cui compare il contatto c_i di nome n e lo cancella, riducendo di uno il numero dei contatti e spostando tutti i contatti successivi indietro di uno. Per esempio, supponiamo che R={contatti[0], contatti[1], contatti[2]} sia una rubrica piena, con tre posti occupati dagli oggetti (contatti) c_0, c_1, c_2. In questo caso numContatti=3. Supponiamo di cancellare il contatto 0. Allora assegnamo numContatti=2. Inoltre assegnamo contatto[0] = contatto[1] e poi contatto[1] = contatto[2], dunque adesso c_0 è sovrascritto, contatto[0] vale c_1, e contatto[1] vale c_2, e contatto[2] vale c_2. Sembra quindi che abbiamo creato un alias, dato che contatto[1] e contatto[2] contengono lo stesso indirizzo c_2. ***Invece non è così***. La variabile contatto[2] è ora irragiungibile dalla classe Rubrica, perché si trova nella posizione 2, e dato che numContatti=2, la posizione 2 non fa parte delle posizioni utilizzabili dalla Rubrica. È vero che in futuro posso aggiungere un nuovo oggetto c'_2 alla rubrica, riportando numContatti a 3. Ma in questo caso contatto[2] diventa c'_2, l'indirizzo di un nuovo oggetto, e l'indirizzo c_2 in contatto[2] viene sovrascritto. In nessun caso il secondo indirizzo c_2 contenuto in contatto[2] può venir utilizzato da un programma che usa classe Rubrica, quindi questo secondo indirizzo non costituisce un alias.

Lezione 07 Diagrammi UML e vettori estendibili

Lezione 07. Parte 1 *(20 minuti)*. La descrizione delle classi con diagrammi.

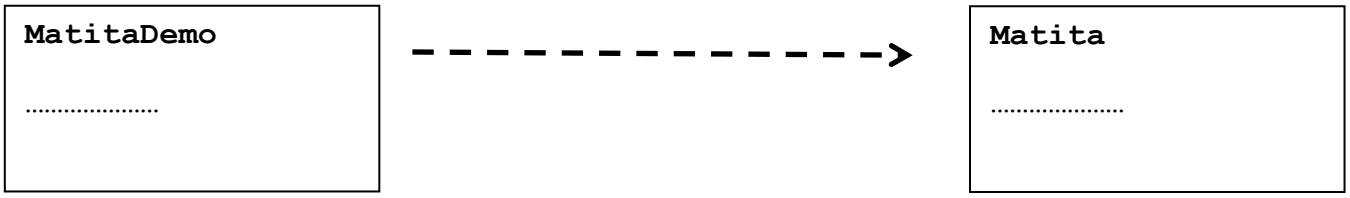
Scrivere programmi è un lavoro di gruppi e non di singoli, e lo stesso programma viene ripreso e modificato da persone diverse. Questo comporta la necessità di fissare un linguaggio comune per descrivere le caratteristiche generali dei programmi. Per i linguaggi a oggetti il linguaggio più comunemente usato si chiama **UML** (**U**nified **M**odeling **L**anguage). Non preoccupatevi, ***non*** vi chiediamo di saper scrivere diagrammi UML all'esame, ma talvolta useremo diagrammi UML a lezione per descrivere delle classi.

Classi in UML. Una classe viene descritta in UML con un rettangolo con tre sezioni orizzontali, il nome della classe, i suoi attributi seguiti da : e il loro tipo, e i metodi, seguiti dalla lista dei loro argomenti e da : e il loro tipo. I metodi pubblici hanno un **+**, i metodi privati un **–**, e i metodi statici sono sottolineati.

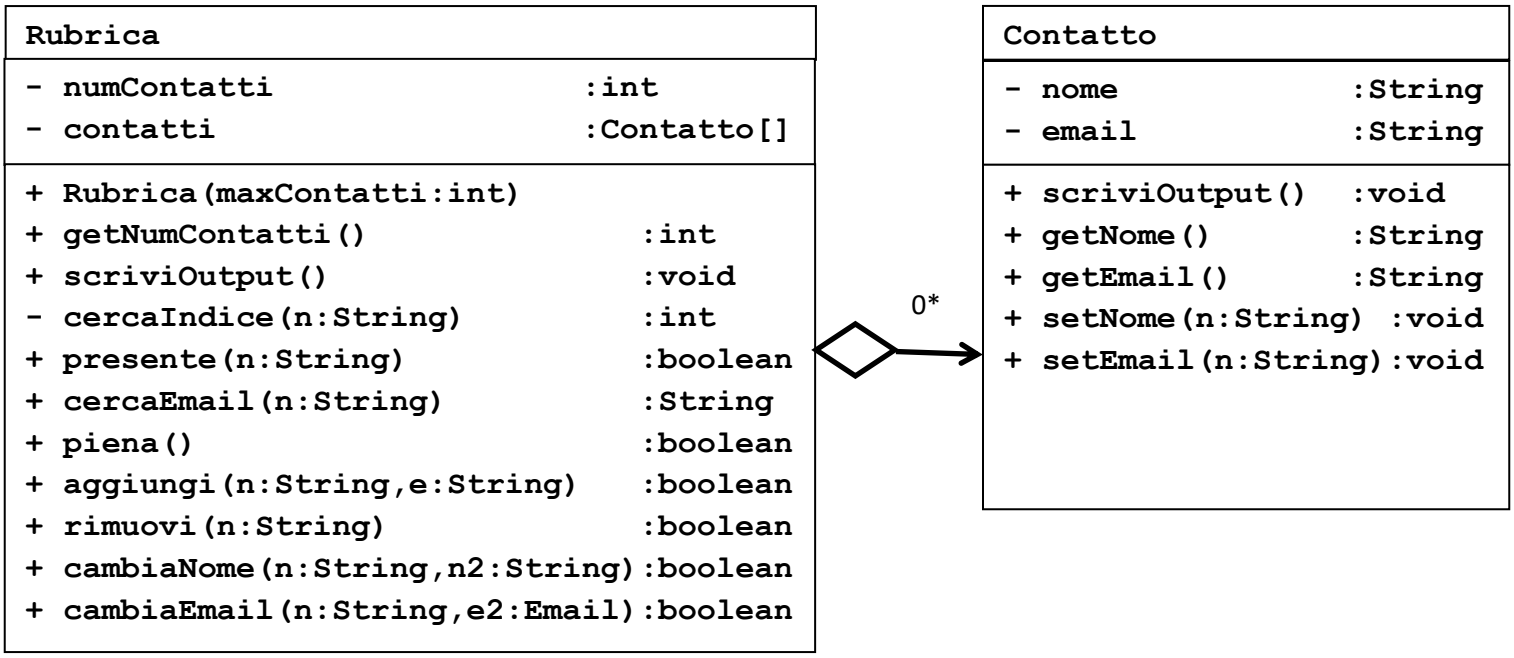
Matita
+ minStelo :int **+ maxStelo :int** **+ maxPunta :int** **- stelo :int** **- punta :int**
+ Matita(stelo:int) **+ disegna() :void** **+ tempera() :void** **+ getStelo() :int** **+ getPunta() :int**

Nella tabella vi sono elencati i metodi riguardanti la classe matita

Una freccia tratteggiata da una classe C a una classe D indica che C ha bisogno di D per la sua definizione e le modifiche di D si ripercuotono su C. Questa relazione viene detta di ***dipendenza***. Per esempio la definizione di MatitaDemo (vedi Esercitazione 01) dipende dalla definizione di Matita:



Una relazione più stretta tra classi è l'***aggregazione***: C aggrega la classe D se gli oggetti di C hanno tra le loro parti gli oggetti di D. L'aggregazione si indica con una freccia che inizia con una losanga bianca. Si può aggiungere un numero o un intervallo di numeri per indicare quante volte un oggetto di D può comparire dentro un oggetto di C. Come esempio, la definizione di Rubrica (vedi Lezione 06) aggrega la definizione di Contatto. Con l'annotazione 0* indichiamo che una rubrica può contenere un numero qualsiasi di contatti (anche zero se è vuota).



Nelle tabelle vi sono elencati i metodi riguardanti le classi rubrica e contatto

Nel caso la classe D compaia ***solo*** come parte della classe C, diciamo che D è parte della ***composizione*** di C e indichiamo la relazione con una freccia che parte da una losanga nera:

Infine, possiamo usare semplici righe per indicare relazioni qualunque tra due classi C e D. In questo caso dobbiamo aggiungere sopra la riga il nome e la direzione della relazione tra C e D.

Lezione 07. Parte 2. Vettori estensibili *(70 minuti)*.. Definiamo una classe di vettori parzialmente riempiti, con la possibilità di aggiungere e togliere elementi. Il vettore raddoppia di dimensioni quando viene chiesto di aggiugere un elemento a un vettore già pieno. In questo caso, gli elementi già esistenti vengono ricopiati nel nuovo vettore, l'indirizzo del vecchio vettore viene sovrascritto con l'indirizzo del nuovo vettore, e il vecchio vettore diventa irraggiungibile.

La classe ArrayExt. Abbiamo un attributo size che all'inizio vale 0. L'***invariante*** di classe è ***(0≤size≤lunghezza vettore)***. Il vettore è riempito nelle posizioni 0, …, size-1. Il costruttore sceglie la lunghezza iniziale del vettore e pone size=0. Abbiamo metodi pubblici per: ***(i)*** ottenere size, ***(ii)*** stampare il vettore, ***(iii)*** ottenere l'elemento di posto 0≤i<size, ***(iv)*** assegnarlo. ***(v)*** Possiamo inserire un elemento in posizione *0≤i≤size (size incluso)* spostando avanti di 1 tutti i successivi, e aggiungendo 1 a size. ***(vi)*** Possiamo rimuovere un elemento in posizione *0≤i<size (size escluso)* spostando indietro di 1 tutti i successivi, e togliendo 1 a size. L'elemento rimosso viene restituito come risultato. ***(vii)*** Il raddoppio del vettore è un metodo privato, utilizzato dalla classe quando aggiungere un elemento fa superare a size la lunghezza del vettore.

Diagramma UML di ArrayExt

ArrayExt	
- size **- vett**	**:int** **:int[]**
+ ArrayExt(min:int) **+ getSize()** **+ scriviOutput()** **+ get(i:int)** **+ set(i:int, x:int)** **- extend()** **+ add(i:int, x:int)** **+ remove(i:int)**	 **:int** **:void** **:int** **:void** **:void** **:void** **:int**

Nella tabella vi sono elencati i metodi riguardanti la classe ArrayExt

Spostamento degli elementi di un vettore. Per spostare un segmento di elementi di vettore di un passo in direzione sinistra->destra

dobbiamo assegnare ripetutamente v[j+1]=v[j] (un elemento al successivo) ***muovendo j nella direzione opposta, destra->sinistra***. In altre parole, se abbiamo v[0],v[1], v[2],v[3] e vogliamo inserire x nel posto 1, dobbiamo assegnare: v[4]=v[3], v[3]=v[2], v[2]=v[1], v[1]=x, muovendo un indice j=3,2,1 da destra a sinistra, in modo da ottenere v[0], v[1], x, v[2], v[3]. Se invece assegnassimo v[1]=x, v[2]=v[1], v[3]=v[2], v[4]=v[3], muovendoci da sinistra a destra, otterremo: v[0], v[1], x, x, x. Non faremmo altro che ricopiare x più volte.

Per la stessa ragione, dobbiamo realizzare lo spostamento degli elementi del vettore di un passo in direzione destra->sinistra assegnando v[j]=v[j+1] ma ***muovendo j nella direzione opposta, sinistra->destra***. Se abbiamo v[0],v[1], v[2],v[3] e vogliamo rimuovere v[1], dobbiamo assegnare: v[1]=v[2], v[2]=v[3], in modo da ottenere v[0],v[2],v[3].

```
//ArrayExt.java
//Questa classe definisce arrays estendibili con dimensioni
//un valore min deciso inizialmente, oppure il doppio, il quadruplo
//eccetera, a seconda di quanto spazio viene richiesto

public class ArrayExt
{ //Invariante: (0 <= size <= lunghezza vett)
  private int size;   //la parte effettivamente in uso del vettore,
                      //all'inizio vale 0
  private int[] vett; //per ora vett vale null
  public int getSize(){return size;}

/** Se min>0, questo metodo mi costruisce un vettore di min elementi
con size=0. La lunghezza di vett sara' min*(una potenza di 2) */
  public ArrayExt(int min)
  {assert min>0 : "Err: min negativo = " + min;
    size=0;
    vett=new int[min];
    assert 0<=size && size<=vett.length;}

  public void scriviOutput(){
    System.out.println( " size = " + size);
    for(int i=0;i<size;++i)
       System.out.println( " vett[" +i+ "]=" +vett[i]);}
```

```
//Metodo di lettura dell'elemento i con 0<=i<size
  public int get(int i)
  {assert 0<=i && i<size;
    return vett[i];}

//Metodo di scrittura dell'elemento i con 0<=i<size
  public void set(int i, int x)
   {assert 0<=i && i<size;
     vett[i]=x;}

//Metodo per espandere il vettore quando necessario
  private void extend()
  {int[] vett1 = new int[vett.length*2];
//nuovo vettore di lunghezza doppia
    for(int i=0;i<size;++i)
    {vett1[i]=vett[i];} //trascrivo il vecchio vettore nel nuovo
    vett=vett1; //aggiorno l'indirizzo del vettore
    assert 0<=size && size<=vett.length;}

//Metodo per aggiungere un elemento x nel posto 0<=i<=size, spostando
//di una posizione gli elementi a destra di i. Puo' fare da push.
public void add(int i, int x)
   {assert 0<=i && i<=size;
    if (size==vett.length)  //se manca lo spazio
       extend(); //espando il vettore
    assert size<vett.length; //ora lo spazio c'e'
     for(int j=size-1;j>=i;--j) {vett[j+1]=vett[j];}
//sposto avanti di una posizione gli elementi a destra di i
//eseguo le assegnazioni nell'ordine da destra a sinistra
     vett[i]=x; //nello spazio cosi' creato aggiungo x
     ++size; //aggiorno il numero degli elementi
     assert 0<=size && size<=vett.length;}

//Rimozione della posizione 0<=i<size effettivamente nel vettore.
//restituisce l'elemento rimosso e quindi puo' fare da "pop"
  public int remove(int i)
   {assert 0<=i && i<size;
    --size; //aggiorno la dimensione
     int x = vett[i]; //salvo vett[i] in x prima di cancellarlo
     for(int j=i;j<=size-1;++j) {vett[j]=vett[j+1];}
//sposto gli elementi a destra di i indietro di uno
//eseguo le assegnazioni nell'ordine da sinistra a destra
```

```
     return x;}}

//ArrayExtDemo.java
public class ArrayExtDemo
{public static void main(String[] args)
 {ArrayExt a = new ArrayExt(10); //capienza iniziale 10
//Per controllare il metodo extend() aggiungo 12 elementi
  System.out.println("aggiungo i valori x=0,1,...,11 sempre in prima
posizione, ognuno davanti ai valori precedenti");
  System.out.println
  ( "ogni aggiunta sposta avanti di uno gli elementi precedenti");
    int x=0; while (x<12) {a.add(0,x);++x;} a.scriviOutput();

  System.out.println
  ( "Rimuovo a[0]=11 e sposto indietro di uno gli altri elementi");
  System.out.println( " a.remove(0)=" + a.remove(0));
  a.scriviOutput();

  System.out.println( "aggiungo x=-1 in posizione 11 (in fondo)");
  a.add(11,-1); a.scriviOutput();}}
```

Nota. Tutte costruzioni appena viste sono di base. Noi le abbiamo svolte come esercizio, ma, volendo cercare, di solito si trovano già fatte nelle librerie. Per esempio, il metodo per ricopiare un vettore in un nuovo vettore esiste nella libreria ***Arrays***, è un metodo statico e si chiama ***copyOf***. Nel caso che ci interessa, ha tipo

int [] copyOf(int[] original, int newLength)

"copyOf" prende un vettore "original" e restituisce una copia di lunghezza minore (che viene troncata) oppure una copia di lunghezza maggiore (che viene estesa con valori di default). Per definire extend() dunque basta aggiungere prima della classe ArrayExt:

import java.util.Arrays;

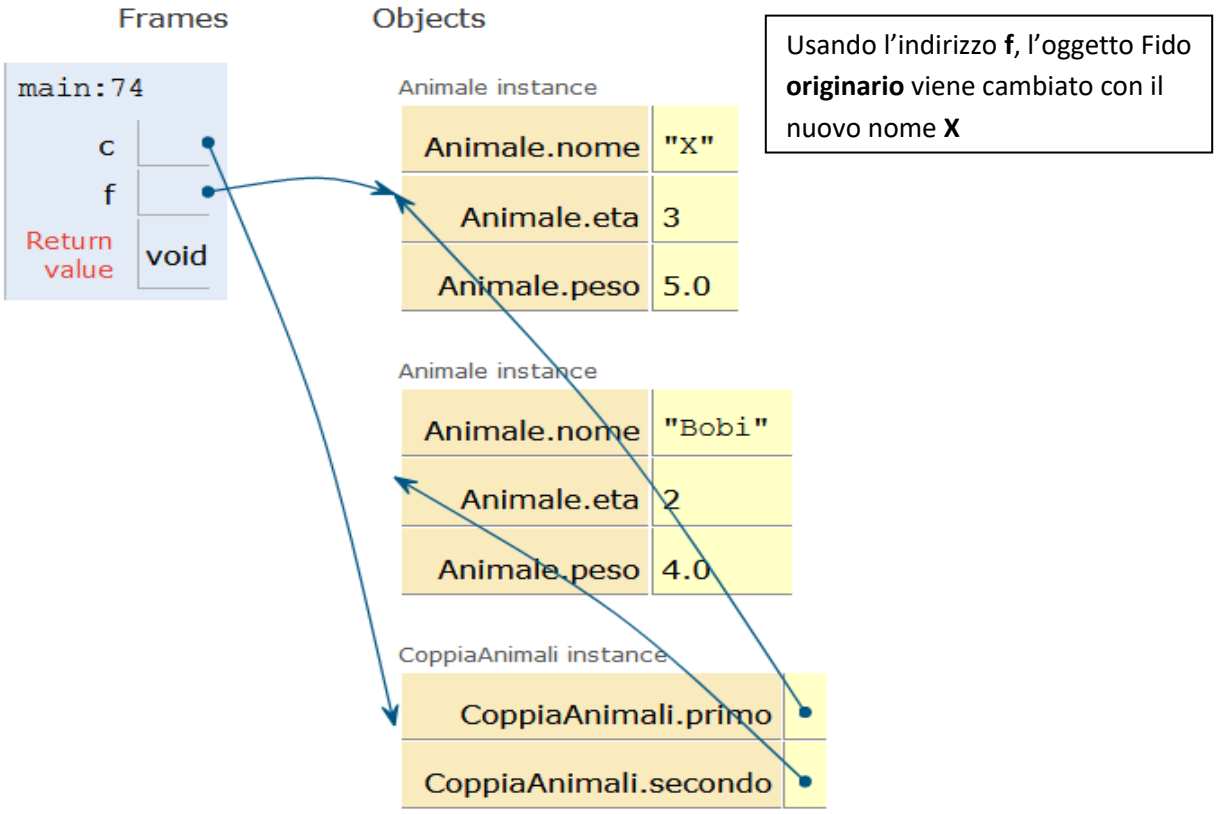
e rimpiazzare la definizione di extend() con:

private void extend(){vett=Arrays.copyOf(vett,2*vett.length);}

Lezione 08 Security Leak, le classi Node e DynamicStack

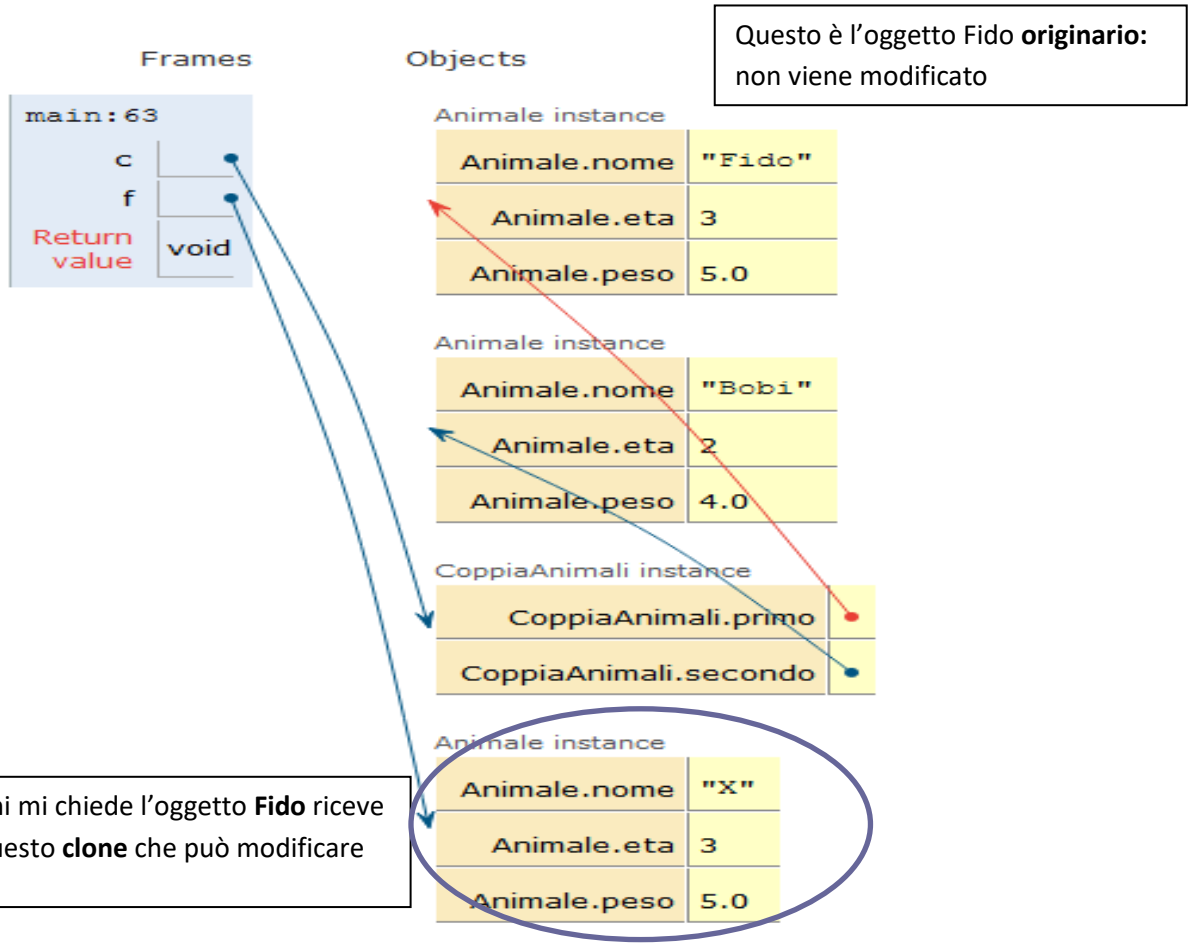
Lezione 08. Parte 1 *(40 minuti)*. Un esempio di *Security Leak*:la classe ***Hacker***. In programmazione dinamica, non è semplice impedire accessi indebiti in lettura/scrittura ai dati. Non basta eliminare i metodi set per impedire gli accessi in scrittura a un dato. Se riesco a trovare un altro percorso al dato tramite i puntatori contenuti nella memoria dinamica posso ancora modificare il dato.

Come esempio, definiamo una classe ***CoppieAnimali*** consentendo solo l'accesso ***get*** (in lettura) ai due oggetti-animali che costituiscono la coppia, senza accesso ***set*** (in scrittura). Vediamo come prendere una coppia di animali Fido e Bobi, e cambiare il nome di "Fido" ***con un nuovo nome "X"***, aggirando il divieto di scrittura posto in CoppieAnimali. Facciamo come segue. Il metodo get di CoppieAnimali non mi consente di modificare l'indirizzo dell'oggetto Fido originario, però mi fornisce una copia **f** di questo indirizzo. Mi basta passare questa copia f a un metodo set (di scrittura) della classe Animali per raggiungere in scrittura i dati di Fido.



Ottenendo una copia f dell'indirizzo di Fido posso modificare Fido scavalcando i divieti di scrittura posti dalla classe CoppiaAnimali

L ’unico modo sicuro per non consentire di modificare gli attributi dei due animali dall'esterno, è passare a richiesta l'indirizzo di una copia o *"clone"* di un oggetto-animale, non l'indirizzo dell’oggetto originario.



Invece fornendo un clone di Fido a chi mi chiede Fido posso proteggere da modifiche l’oggetto originale Fido

Vediamo ora in dettaglio come definire le classi Animale e CoppiaAnimali, e come violare i divieti di scrittura posti in CoppiaAnimali.

```
//Classe Animale: animali di cui e' noto nome, eta' e peso
//abbiamo un costruttore, metodi get e set, metodo di stampa
public class Animale
{ private String nome;
  private int eta;
  private double peso;
```

```
  public String   getNome()          {return nome;}
  public int      getEta()           {return eta;}
  public double   getPeso()          {return peso;}
  public void     setNome(String n)  {nome=n;}
  public void     setEta(int e)      {eta=e;}
  public void     setPeso(double p)  {peso=p;}

  public Animale(String n, int e, double p)
   {nome=n;eta=e;peso=p;}

public void scriviOutput()
{System.out.println
(" nome=" + nome + " eta=" + eta + " peso=" + peso);}}


//CoppiaAnimali.java
public class CoppiaAnimali
{private Animale primo; private Animale secondo;
//Una coppia di animali nasce come la coppia primo=null, secondo=null
//All'inizio primo, secondo non sono valori corretti di tipo Animale

 public CoppiaAnimali
/** Ho bisogno di "new" per ottenere valori corretti per primo,
secondo. Invece scrivere primo.setNome(n1); dato che primo nasce
"null" produce a run time una null pointer exception */
 (String n1, int e1, double p1, String n2, int e2, double p2)
 {primo=new Animale(n1,e1,p1); secondo=new Animale(n2,e2,p2);}

public    Animale    getPrimo()   {return primo;}
public    Animale    getSecondo() {return secondo;}

/**Se non vogliamo consentire di modificare gli attributi dei due
animali dall'esterno, dobbiamo passare l'indirizzo di una copia o
"clone" dei due animali, non l'indirizzo originario */
/** Metodi di clonazione:
public    Animale    getPrimo()
{return   new Animale
  (primo.getNome(), primo.getEta(), primo.getPeso());}
public    Animale    getSecondo()
{return   new Animale
  (secondo.getNome(), secondo.getEta(), secondo.getPeso());} */
```

```
  public void scriviOutput()
  {primo.scriviOutput(); secondo.scriviOutput();}}

//Hacker.java (Vediamo come aggirare i divieti di scrittura)
public class Hacker
{public static void main(String[] args)
 {System.out.println("Definisco Fido e Bobi");
  CoppiaAnimali c = new CoppiaAnimali("Fido",3,5.0,"Bobi",2,4.0);
  c.scriviOutput();
  System.out.println("Chiedo una copia dell'indirizzo di Fido");
  Animale f = c.getPrimo();
  System.out.println
  ("Anche con una copia dell'indirizzo posso modificare Fido");
  f.setNome("X");
  c.scriviOutput();
  System.out.println
  ("Invece non si puo' modificare Fido se getPrimo da' un clone");}}
```

Lezione 08. Parte 2. Pile dinamiche *(50 minuti)*.

Un nodo è la coppia di un dato (nel nostro caso: un intero) e di un puntatore a un nodo. Si tratta quindi di una definizione *ricorsiva*. C'è un nodo speciale, "null", che non contiene nulla, e che fa da punto di partenza, ogni altro nodo punta a un altro nodo. Usando i nodi costruiremo per esempio le pile.

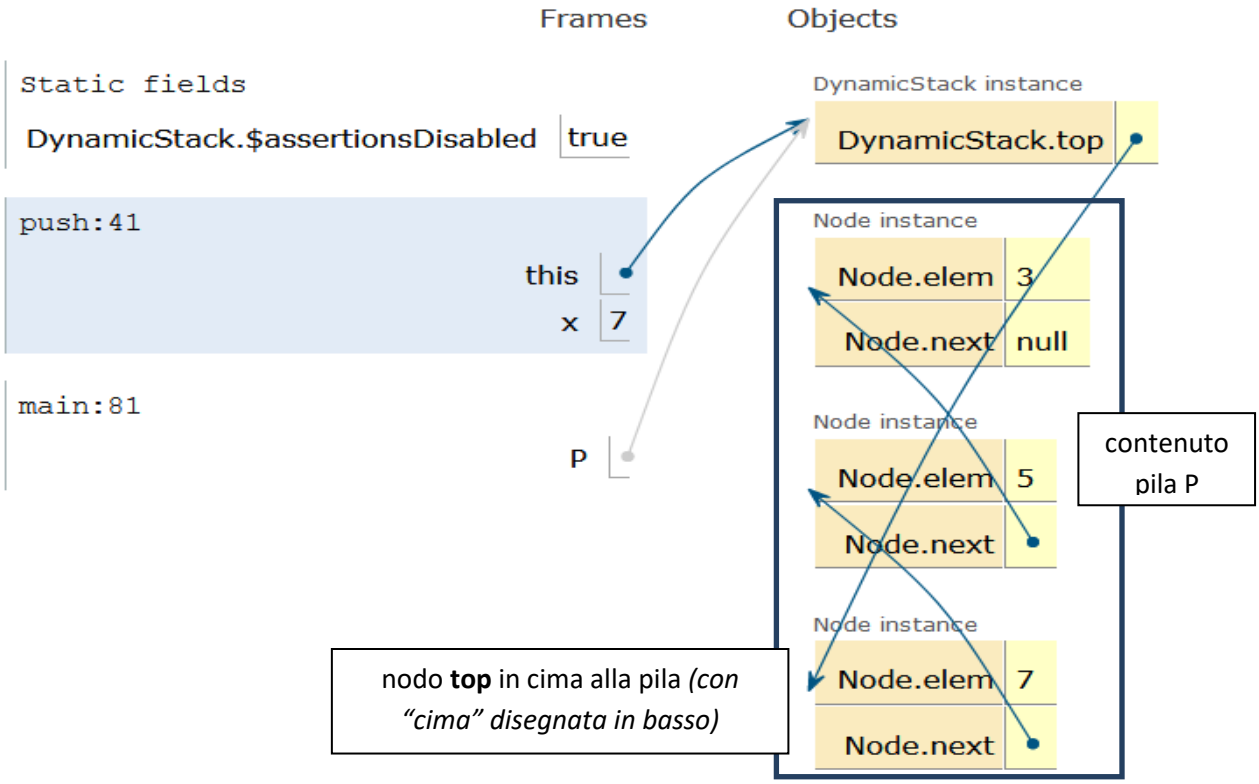
La classe Nodo ha attributi privati un elemento di tipo int e un puntatore next a un altro nodo, un costruttore che richiede un elemento e un puntatore per costruire un nodo, e tutti i metodi get e set.

```
//Node.java
public class Node
{ private int elem;
  private Node next;
  public Node(int elem, Node next){this.elem=elem;this.next=next;}

  public int getElem(){return elem;}
  public Node getNext(){return next;}
  public void setElem(int elem){this.elem=elem;}
  public void setNext(Node next){this.next=next;}}
```

Vediamo ora come utilizzare i nodi per realizzare ***la classe DinamicStack*** delle pile completamente dinamiche, che possono

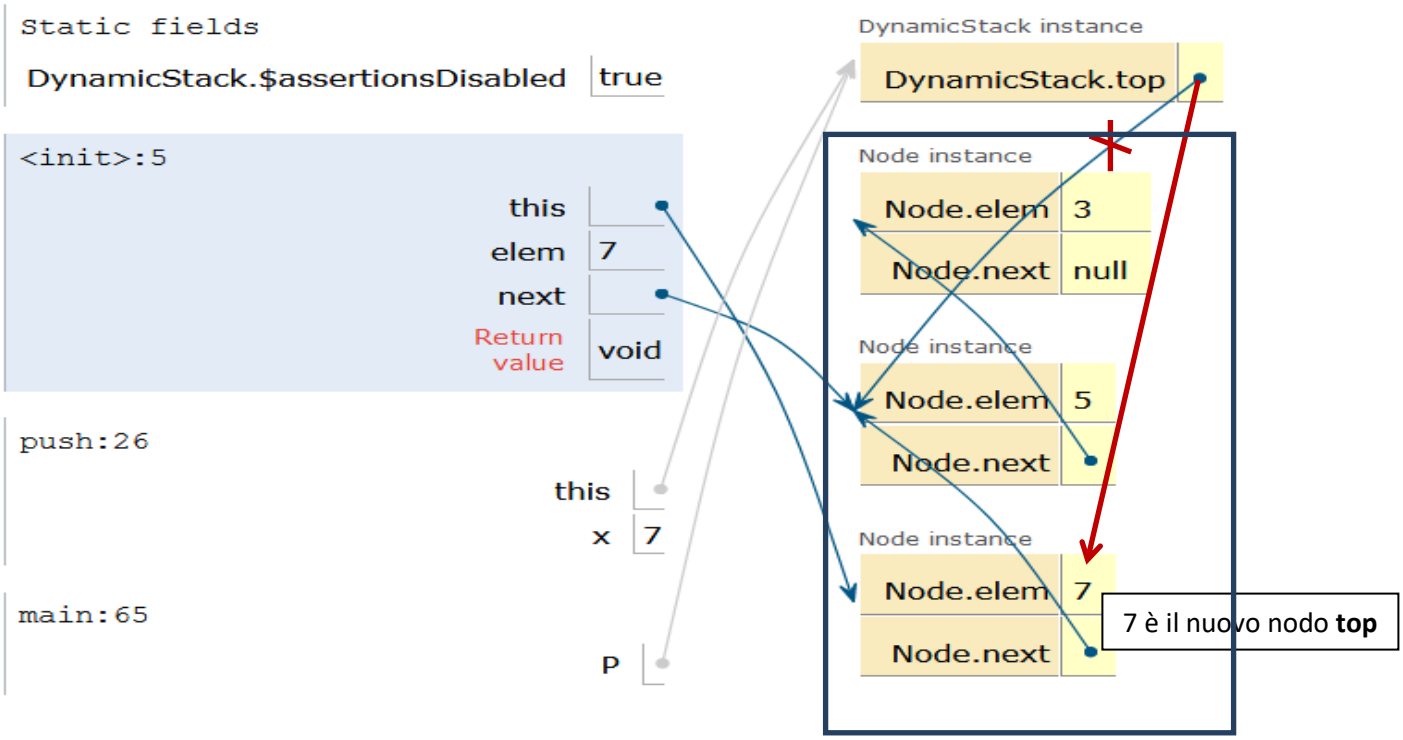
contenere un numero arbitrario di elementi e la cui occupazione di memoria cresce o cala in corrispondenza del numero di elementi in essa contenuti.

Una pila dinamica è fatta di una lista di nodi: n_0, n_1, ...,n_{k-1}. Ogni nodo che contiene un elemento e l'indirizzo del nodo precedente tranne n_0 che contiene l'indirizzo "null". La pila e' identificata con l'indirizzo del nodo n_{k-1}, che viene chiamato "top", e con "null" se k=0 (se la pila è vuota). C'è una operazione ***void push(int x)*** che aggiunge un nodo di elemento x in "cima" a una pila, una operazione ***int pop()*** che toglie il nodo in cima ad una pila non vuota e ne restituisce l'elemento, ***int top()*** che restituisce l'elemento del nodo in "cima" senza eliminarlo, e un test ***boolean empty()*** che controlla se la pila è vuota. Aggiungiamo un metodo di scrittura e dei costruttori. Come esempio, vediamo la disposizione nella memoria per la pila **P={3,5,7}** (qui disegnata con la "cima" in basso).



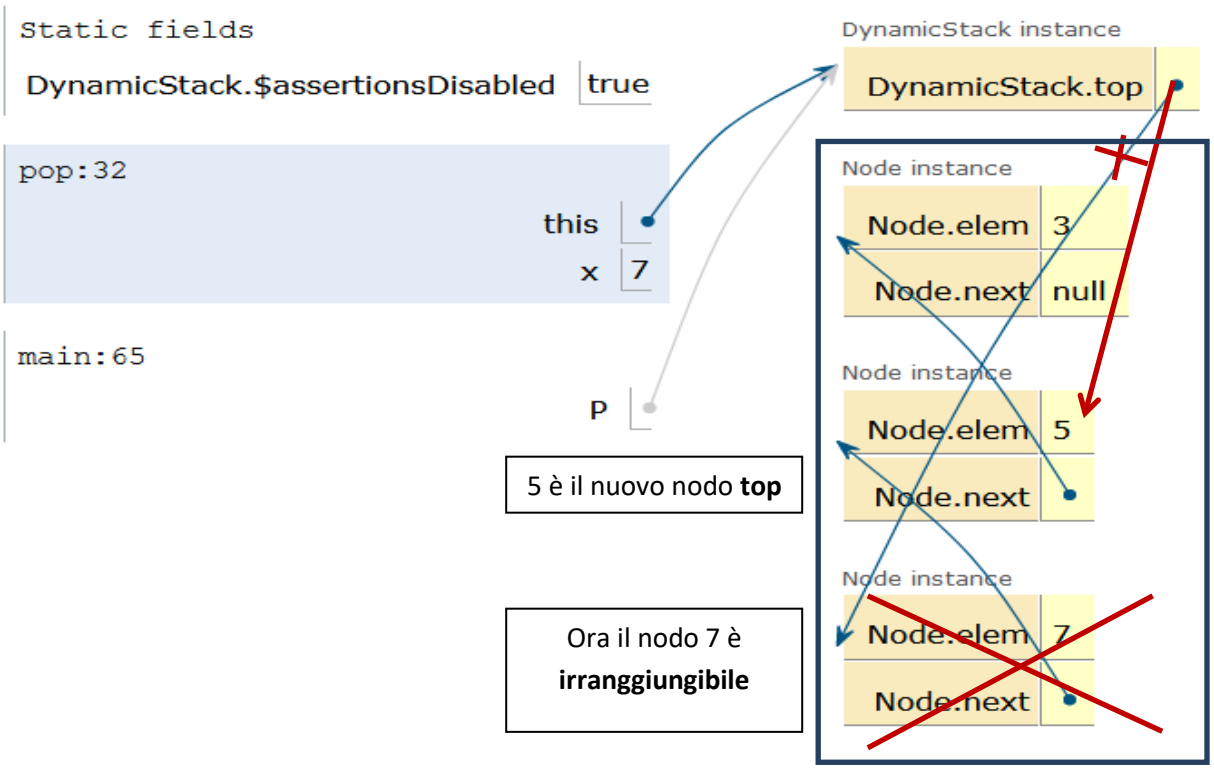
La pila P={3,5,7}

Vediamo come ottenere la pila **P={3,5,7}** a partire dalla pila **P={3,5}** aggiungendo 7 con un ***push***. Il nuovo nodo 7 punta al nodo 5, il vecchio valore di "top", e "top" viene riassegnato al nodo 7.



La pila P={3,5,7} ottenuta dalla pila P={3,5} con un "push"

Eliminiamo 7 usando un pop: ora "top" torna a puntare a 5, e il nodo 7 non è più raggiungibile, il suo spazio di memoria verrà riciclato.



Nell'immagine vi la rappresentazione della Pila con l'elemento 5 diventato nodo top e di conseguenza l'elemento 7 è irraggiungibile

Realizziamo ora la classe ***DynamicStack***. Tutti i metodi di DynamicStack devono preservare il seguente ***invariante della classe***: ogni nodo tranne il primo punta al ***precedente***, e top punta al primo elemento della pila. Come già visto per i vettori estendibili, non consentiamo nessun accesso diretto ai nodi della pila: ogni accesso ai dati nella pila avviene con le operazioni della pila.

```
//DynamickStack.java
public class DynamicStack{

private Node top;
//ultimo nodo aggiunto alla pila. "null" se non ce ne sono

//COSTRUTTORE di una pila P = {} vuota
public DynamicStack(){top = null;}

//test se la pila e' vuota
public boolean empty(){return top==null;}

//aggiungo un nodo in cima alla pila con un nuovo elemento x
```

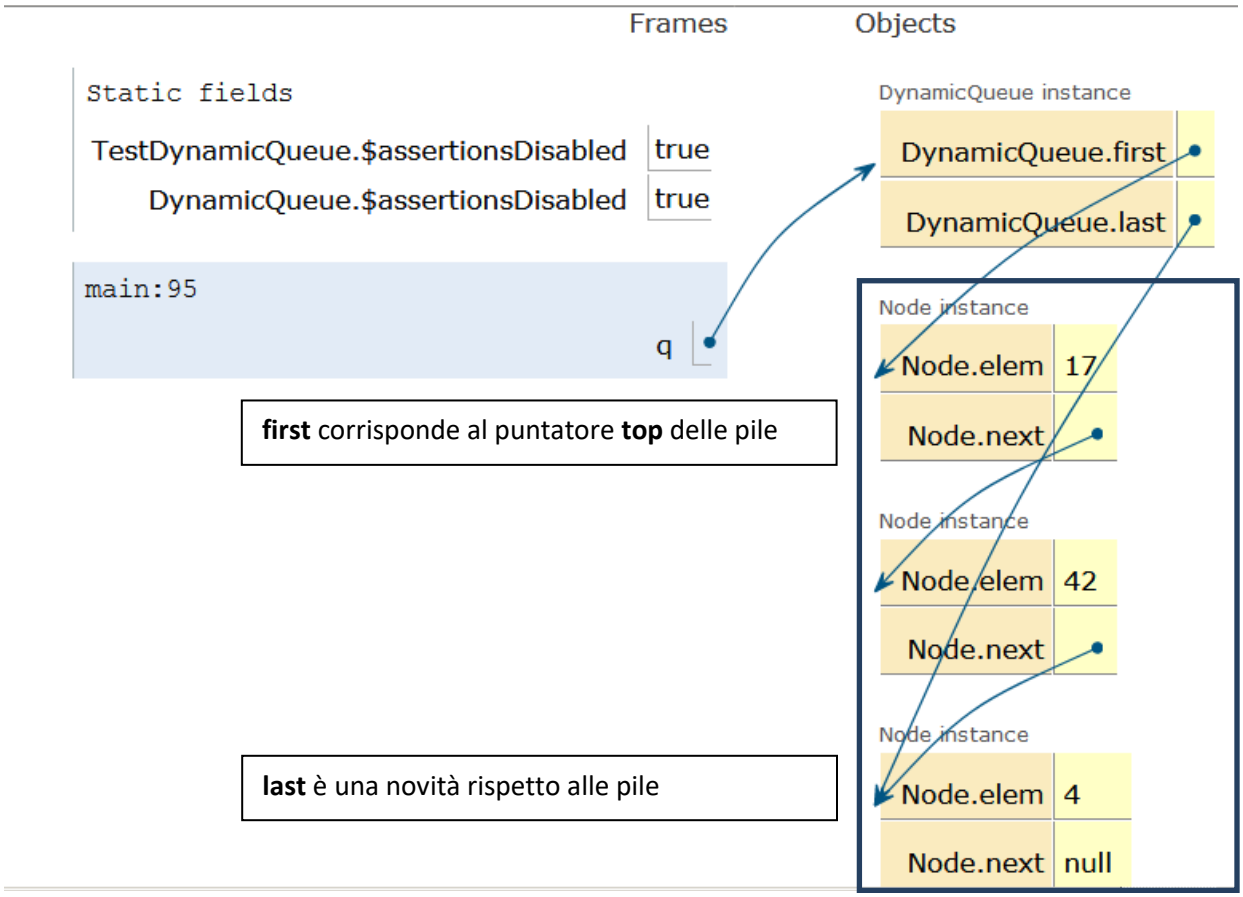
```
public void push(int x) {top = new Node(x,top);}

//tolgo il nodo in cima alla pila e restituisco il suo contenuto
public int pop()
{assert !empty();
 int x = top.getElem();
 top = top.getNext(); //elimino l'ultimo nodo con contenuto x
 return x;}

//restituisco il contenuto del nodo in cima alla pila senza toglierlo
public int top()
{assert !empty();
 int x = top.getElem();
 return x;}

//STAMPA. Per scorrere una pila usiamo una variabile di tipo nodo
//che parte da top e procede all'indietro lungo la pila fino
//ad arrivare al nodo null

  public void scriviOutput()
  {Node temp = top; //partiamo dal nodo in cima alla pila
   while (temp != null) //ci fermiamo quando temp arriva al nodo null
   {System.out.println( " || " + temp.getElem());
    temp=temp.getNext(); //arretriamo al nodo precedente
   }}


//COSTRUTTORE di una pila P = {1,...,n}, pila vuota se n<=0
public DynamicStack(int n)
{top = null; int i = 1;
 while (i<=n) //aggiungo il nodo che contiene i
 {top = new Node(i,top);i++;}}
}

//DynamicStackDemo.java (prova della classe DynamicStack)
public class DynamicStackDemo
{public static void main(String[] args)
 {System.out.println( "Stampo la pila P = {3,5,7,9,11}");
  DynamicStack P = new DynamicStack();
  P.push(3);P.push(5);P.push(7);P.push(9);P.push(11);
  P.scriviOutput();
```

```
  System.out.println( "Estraggo gli ultimi 3 elementi inseriti: 11,
9, 7. Leggo 5");
  System.out.println(P.pop());
  System.out.println(P.pop());
  System.out.println(P.pop());
//Leggiamo il prossimo elemento, 5, senza estrarlo dalla pila
  System.out.println(P.top());
  System.out.println( "Stampo cosa resta: P={3,5}" );
  P.scriviOutput();
  System.out.println( "Stampo Q={1,2,3,4,5,6,7,8,9,10}" );
  DynamicStack Q = new DynamicStack(10);
  Q.scriviOutput();}}
```

Esercitazione 02 Code dinamiche

Una coda è una struttura dati in in cui gli elementi vengono inseriti/rimossi secondo la politica ***FIFO (First-In-First-Out):*** il primo elemento inserito è il primo a essere rimosso. Una coda viene usata per eseguire dei compiti nello stesso ordine con cui si presentano.

Vi chiediamo di definire una implementazione ***DynamicQueue*** delle code dinamiche usando la classe di nodi vista in precedenza, e adattando l'implementazione delle pile dinamiche vista nella lezione precedente. Una coda dinamica viene definita come una lista di nodi (anche vuota) in cui ogni nome punta al ***precedente*** (come nella pila) con due attributi privati: un puntatore ***first*** al primo elemento della coda (il primo ad essere eliminato) e un puntatore ***last*** all'ultimo elemento della coda, l'ultimo arrivato, dietro al quale aggiungeremo il prossimo elemento. Potete immaginare una coda dinamica come una pila dinamica dove "top" viene chiamato "first" e dove abbiamo un nuovo puntatore, "last". Qui disegnamo il "first" in alto.



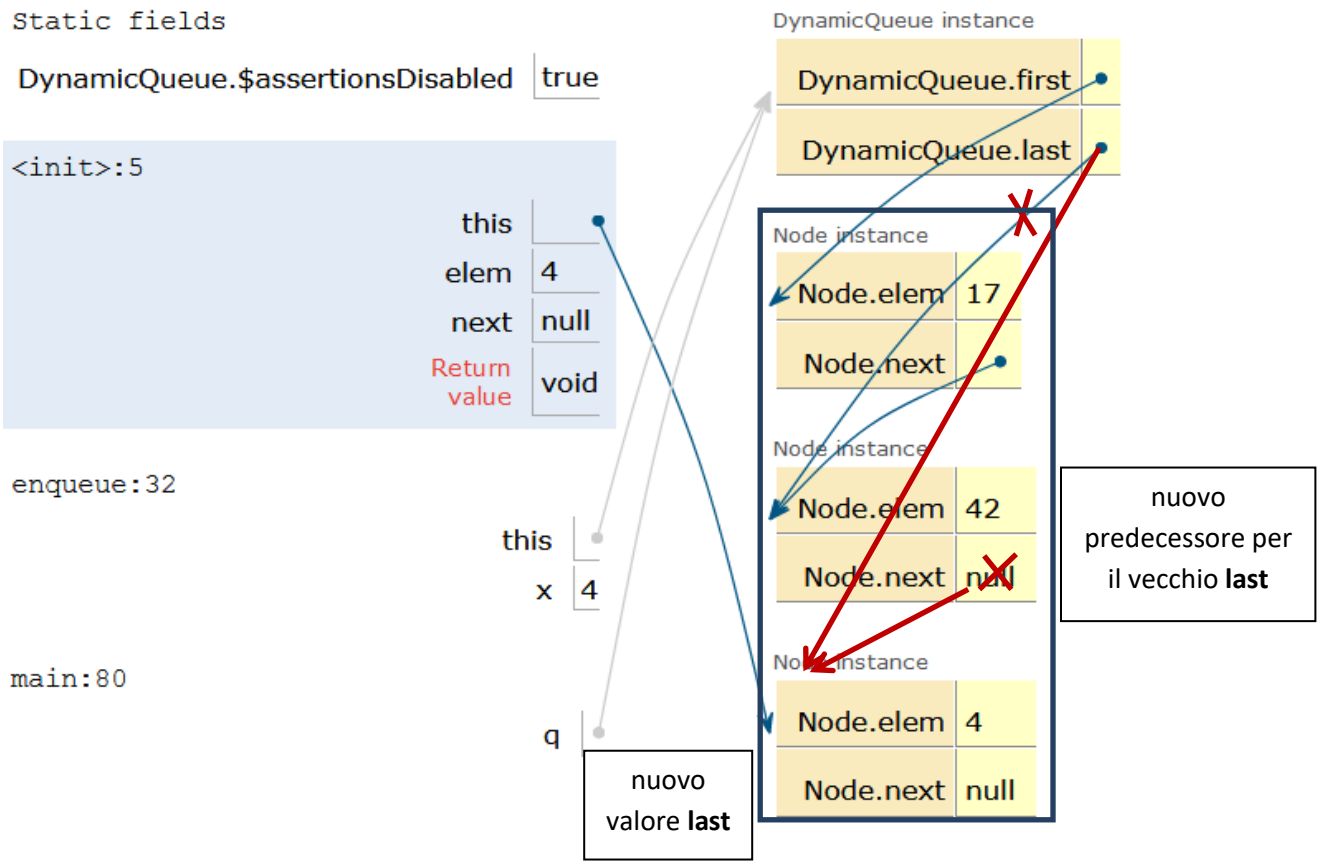
La coda q={17,42,4}: "first" punta a 17 e "last" punta a 4

Tutti i metodi di **DynamicQueue** sono pubblici e dinamici. Definite ***(i)*** un costruttore per la coda vuota, ***(ii)*** un metodo di scrittura, ***(iii)*** un metodo ***void enqueue(int x)*** per aggiungere un elemento dietro l'ultimo, ***(iv)*** un metodo ***int dequeue()*** per togliere il primo elemento della coda, ***(v)*** un metodo ***int size()*** per contare gli elementi della coda, ***(vi)*** un metodo ***int front()*** per leggere il primo elemento della coda senza toglierlo ***(vii)*** un metodo ***boolean empty()*** per verificare se la coda è vuota.

Suggerimento. Definite i cicli dentro i metodi di di **DynamicQueue** usando come come indice l'indirizzo di un nodo p. **Facoltativo.** Definite un metodo pubblico ***boolean contains(int x)*** per verificare se la coda contiene un dato elemento x.

Tutti i metodi devono preservare il seguente ***invariante della classe***: ogni nodo tranne l'ultimo punta al ***precedente***, e first e last puntano al primo e all'ultimo elemento della coda, sono uguali a ***null*** se la coda è vuota.

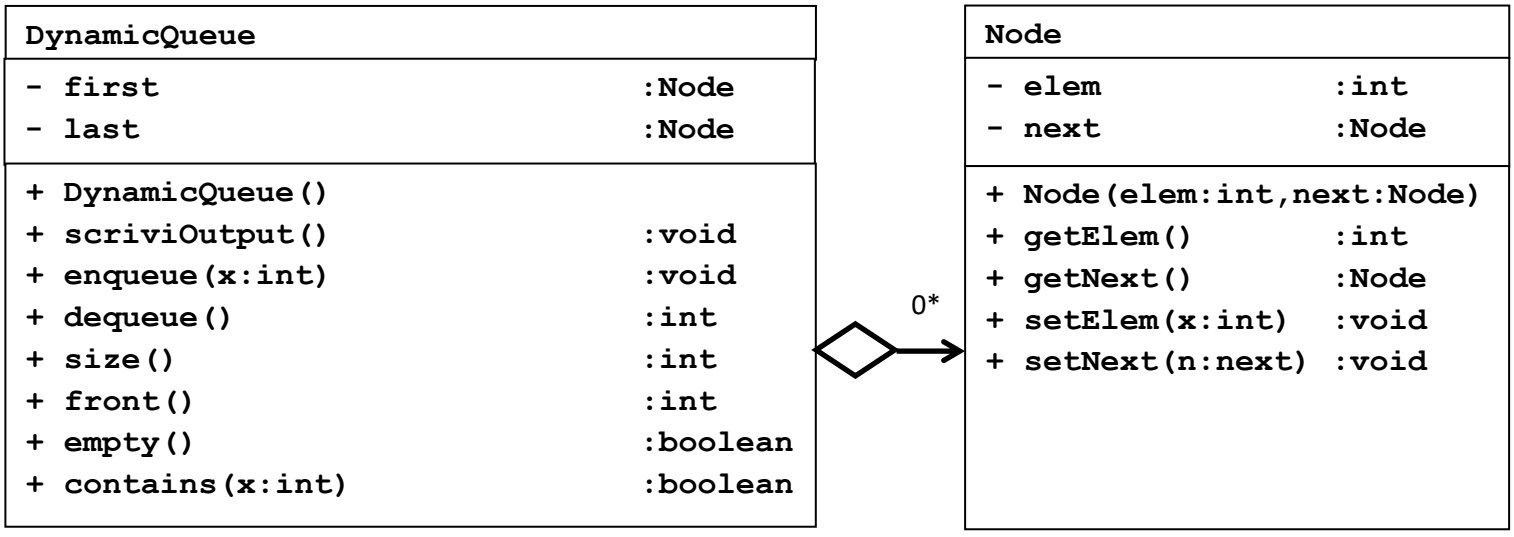
Esecuzione q.enqueue(4) con q={17,42}



Nell'immagine vi è rappresentata una pila statica con i vari controlli tramite asserzioni

Diagramma UML per le code dinamiche

Una coda dinamica è definita ***aggregando*** 0 o più elementi della classe Node.



Usate la classe TestDynamicQueue inclusa qui sotto come test per la classe DynamicQueue.

```
//Node.java
//Riutilizzate la classe Node definita nella Lezione 08

//TestDynamicQueue.java
//Usate questa classe come test per DynamicQueue
public class TestDynamicQueue
{public static void main(String[] args){
 DynamicQueue q = new DynamicQueue();
 System.out.println( "q = {17,42,4} " );
 q.enqueue(17); q.enqueue(42); q.enqueue(4);
 q.scriviOutput();
 System.out.println( "q.empty() = " + q.empty());
/** Aggiungete queste righe se avete realizzato "contains"
 System.out.println( "q.contains(4)  = " + q.contains(4)); //true
 System.out.println( "q.contains(40) = " + q.contains(40));//false
*/
 System.out.println("q.size() = " + q.size());  // stampa 3
 System.out.println("q.front()= " + q.front()); // stampa 17
 System.out.println(q.dequeue()); //toglie e stampa 17
 System.out.println(q.dequeue()); //toglie e stampa 42
 System.out.println(q.dequeue()); //toglie e stampa 4: coda vuota
 // gli elementi vengono stampati nello stesso ordine in cui
 // sono stati inseriti, dal momento che la coda e' una
 // struttura FIFO (First-In-First-Out)
 System.out.println( "q.empty() = " + q.empty());
 /** Questo comando deve far scattare un "assert":
 q.front();
 */
    }
}
```

Soluzione Esercitazione 02 del 2019

```
//DinamicQueue.java
public class DynamicQueue {
    private Node first;
    private Node last;

    public DynamicQueue()
    {
 first = last = null;
    }

public void scriviOutput()
  {Node temp = first; //partiamo dal primo nodo della coda
   while (temp != null) //ci fermiamo quando temp arriva al nodo null
   {System.out.println( " " + temp.getElem());
    temp=temp.getNext(); //arretriamo al nodo precedente
   }}

    // inserimento di un elemento in fondo alla coda
    public void enqueue(int x)
    {
 Node node = new Node(x, null);
 if (first == null)
     first = last = node;
 else {
     last.setNext(node);
     last = node;
 }
    }

    // rimozione del first elemento della coda
public int dequeue()
    {
 assert !empty(): "Err. rimozione da coda vuota.";
 int x = first.getElem();
 first = first.getNext();
 if (first == null) last = null;
 return x;
    }

  // calcolo della dimensione della coda
```

```
    public int size()
    {
 int n = 0;
 for (Node p = first; p != null; p = p.getNext())
     n++;
 return n;
    }

// test per verificare se la coda e' vuota
public boolean empty(){return first == null;}

// metodo per leggere senza rimuovere il primo elemento della coda
public int front()
{assert !empty(): "Err. lettura da coda vuota.";
 return first.getElem();}

// test per verificare se la lista p contiene x
 public boolean contains(int x)
 {Node p=last;
  while (p!=null)
  {if (p.getElem()==x) return true; else p=p.getNext();}
  return false;}

}
```

Lezione 09 Metodi statici per la classe Node

Lezione 09. La classe NodeUtil (esercizi su liste concatenate). Facciamo riferimento alla classe ***Node*** introdotta nella Lezione 08. Chiamiamo ***"lista concatenata"*** l'insieme dei nodi raggiungibili da un nodo dato seguendo il puntatore "next" del nodo, nel caso in cui questo percorso non contenga cicli e termini in un puntatore ***null***. Una lista concatenata si rappresenta con il nodo da cui partiamo per percorrerla, nodo che chiamiamo ***il nodo in cima*** alla lista. Chiamiamo il nodo a cui punta un nodo dato ***il nodo precedente*** nella lista. Abbiamo già visto come definire pile e code a partire da liste.

Definiamo una classe ***NodeUtil*** con metodi ***pubblici e statici*** sulle liste concatenate: sono simili a metodi già visti a ProgI per i vettori. Le differenze riguardano l'uso di indirizzi al posto di indici: ***(i)*** usiamo ***null*** al posto della posizione di indice -1 (quella a sinistra della prima posizione realmente esistente) ***(ii)*** usiamo il nodo in cima alla lista al posto della posizione ***(lungh. vettore-1)*** ***(iii)*** usiamo un nodo p al posto di un indice i per indicare una posizione della lista ***(iv)*** usiamo l'assegnazione p = p.getElem() al posto dell'assegnazione i=i-1 per ottenere la posizione precedente a una posizione data nella lista.

Qui solo l'elenco dei metodi che vogliamo definire. ***Vi consigliamo di ripetere gli stessi esercizi a casa.*** Le soluzioni viste in classe sono subito dopo. Per i metodi 1-4 daremo anche una ***soluzione ricorsiva***. Per i metodi 5-7 non indichiamo suggerimenti.

0. **void scriviOutput(Node p).** Stampa una lista concatenata partendo dal nodo in cima alla lista andando indietro. *Adattate il metodo per stampare una pila della Lezione 08.*
1. **int length(Node p).** Calcola la lunghezza di una lista. Traversiamo la lista con un ciclo, aggiungendo uno ogni volta che troviamo un nodo non nullo.
2. **int sum(Node p).** Somma degli elementi di una lista. Traversiamo la lista con un ciclo, sommando tutti gli elementi che incontriamo e mantenendo il risultato in una variabile s. Finita la lista, s è la somma di tutti gli elementi della lista.
3. **int max(Node p).** Massimo degli elementi di una lista non nulla (non definito per la lista vuota). Traversiamo la lista con un

ciclo, mantendo in una variabile m il più grande degli elementi trovati. Alla fine della lista m è il massimo.

4. **boolean member(Node p, int x).** Controlla se x dato compare in una lista p. Traversiamo la lista con un ciclo, e non appena troviamo x usciamo con risposta true. Se arriviamo alla fine della lista senza trovare x, restituiamo false.
5. **String toString(Node p).** Restituisce una stringa con i nodi di p scritti in ordine inverso.
6. **boolean sorted(Node p).** Verifica se una lista concatenata è ordinata in modo debolmente crescente.
7. **booleans equals(Node p, Node q).** Verifica se due liste concatenate sono uguali.

```
//Node.java: riutilizziamo la classe gia' vista nella Lezione 08
public class Node
{ private int elem;
  private Node next;
  public Node(int elem, Node next){this.elem=elem;this.next=next;}

  public int getElem(){return elem;}
  public Node getNext(){return next;}
  public void setElem(int elem){this.elem=elem;}
  public void setNext(Node next){this.next=next;}}
```

Main di prova. **Copiate** questo main al fondo della classe ***NodeUtil*** che definirete per risolvere gli esercizi. Questo main non funziona da solo o se inserito in altre classi.

```
 public static void main(String[] args)
  { System.out.println( "Le stampe sono tutte in ordine inverso");
    System.out.println( "----------------------");
    Node q = new Node(1,null);q = new Node(2,q);q = new Node(3,q);
    q = new Node(4,q);
    System.out.println( "Lista q:");
    scriviOutput(q);
    System.out.println( "----------------------");
    Node p = new Node(3,q); p = new Node(2,p); p = new Node(1,p);
    System.out.println( "Lista p:");
    scriviOutput(p);
    System.out.println( "----------------------");
```

```
        System.out.println( "1. length(p) = "       + length(p));
        System.out.println( "1. length_rec(p) = "   + length_rec(p));
        System.out.println( "---------------------");
        System.out.println( "2. sum(p) = "          + sum(p));
        System.out.println( "2. sum_rec(p) = "      + sum_rec(p));
        System.out.println( "---------------------");
        System.out.println( "3. max(p) = "          + max(p));
        System.out.println( "3. maxr_rec(p) = "     + max_rec(p));
        System.out.println( "---------------------");
        System.out.println( "4. member(p,3) = "     + member(p,3));
        System.out.println( "4. member(p,5) = "     + member(p,5));
        System.out.println( "4. member_rec(p,3) = " + member_rec(p,3));
        System.out.println( "4. member_rec(p,5) = " + member_rec(p,5));
        System.out.println( "---------------------");
        System.out.println( "5. toString(q) = "     + toString(q));
        System.out.println( "5. toString(p) = "     + toString(p));
        System.out.println( "---------------------");
        System.out.println( "6. sorted(q) = "       + sorted(q));
        System.out.println( "6. sorted(p) = "       + sorted(p));
        System.out.println( "---------------------");
        System.out.println( "7. equals(p,q) = "     + equals(p,q));
        System.out.println( "7. equals(p,p) = "     + equals(p,p));
        System.out.println( "7. equals(q,q) = "     + equals(q,q));
        System.out.println( "7. equals(q,p) = "     + equals(q,p));}

// Qui deve esserci la parentesi di chiusura della classe NodeUtil
```

Soluzioni per la Lezione 09.

Queste sono le soluzioni degli esercizi 1-7 della Lezione 09. Per gli esercizi 1-4 sono date anche le soluzioni ricorsive. Copiate al fondo di questa classe il main di prova che avete ricevuto insieme agli esercizi e eseguitelo per controllare le vostre soluzioni.

```
//NodeUtil.java:
public class NodeUtil{
//0. STAMPA dei nodi della lista in ordine inverso (vedi Lez.08)
public static void scriviOutput(Node p)
{while (p!=null){System.out.println(p.getElem());p=p.getNext();}}

//1. Length. Metodo che calcola la lunghezza di una lista.
  public static int length(Node p)
  {int l=0;
    while (p !=null)
//ogni volta che incontro un nodo incremento di 1 la lunghezza
     {p=p.getNext(); l++;}
    return l;
  }

//versione ricorsiva
  public static int length_rec(Node p)
  {if (p==null) return 0; else return 1 + length_rec(p.getNext());
  }

//2. Sum. Somma degli elementi di una lista.
  public static int sum(Node p)
  {int s=0;
    while (p !=null)
//ogni volta che incontro un nodo ne aggiungo il contenuto alla somma
     {s = s+p.getElem(); p=p.getNext();}
    return s;}

//versione ricorsiva
  public static int sum_rec(Node p)
  {if (p==null) return 0; else return p.getElem() +
sum_rec(p.getNext());}

//3. Max. Massimo degli elementi di una lista non nulla
//(non definito per la lista vuota).
```

```
    public static int max(Node p)
  {assert p!= null: "Err. Massimo di una lista vuota";
    int m=p.getElem(); p=p.getNext();
// m=massimo dei nodi gia' visti, all'inizio m=nodo in cima

    while (p !=null)
// a ogni passo prendo il massimo tra m (max nodi gia' visti)
// e il nodo corrente.
       {m = Math.max(m,p.getElem()); p=p.getNext();}

//alla fine m e' il massimo tra tutti i nodi
    return m;}

//versione ricorsiva
  public static int max_rec(Node p)
  {assert p!= null: "Err. Massimo di una lista vuota";
    if (p.getNext()==null) return p.getElem();
    else return Math.max(p.getElem(),max_rec(p.getNext()));
  }

//4. Member. Metodo che controlla se x dato compare in una lista p.
  public static boolean member(Node p, int x)
  {while (p !=null)
//a ogni passo se trovo x restituisco "true"
     {if (p.getElem()==x) return true; else p=p.getNext();}

//se ho esaurito la lista senza trovare x allora x non c'e'
    return false;
  }

//versione ricorsiva
  public static boolean member_rec(Node p, int x)
  {if (p==null) return false;
    else if (p.getElem()==x) return true;
    else return member_rec(p.getNext(),x);
  }

// 5. String toString(Node p) restituisce una stringa
// con i nodi di p scritti in ordine inverso
  public static String toString(Node p)
  {String s = " ";
   while (p!=null){s=s+p.getElem()+" ";p=p.getNext();}
```

```
   return s;}

// 6. Sorted(Node p) verifica se una lista concatenata
// è ordinata in modo debolmente crescente
  public static boolean sorted(Node p)
  {if (p==null) return true; //lista vuota: ordinata
   while (p.getNext()!=null)
   {if (p.getNext().getElem()>p.getElem())
      return false;
//se (penultimo elemento > ultimo elemento): lista non ordinata
     p=p.getNext();}
//finita la lista, non c'e' un elemento > del seguente:lista ordinata
   return true; }

// 7. equals(Node p, Node q) verifica se due liste concatenate
// sono uguali
public static boolean equals(Node p, Node q)
 {while ((p!=null) && (q!=null))
   {if (p.getElem()!=q.getElem()) return false;
//se trovo due elementi in posizioni uguali e diversi: p,q diverse
     p=p.getNext();q=q.getNext();}

//finito il while abbiamo p=null oppure q=null. Quindi:
// se p,q sono lo stesso indirizzo, allora p,q=null sono uguali
// se p,q sono indirizzo diversi, allora uno e' null e l'altro no
   return (p==q);
 }

////////////////////////////////////
//           MAIN DI PROVA
// COPIATE QUI il main di prova dato
// insieme agli esercizi ed eseguitelo
////////////////////////////////////

}
```

Lezione 10 Classi generiche: coppie, nodi e pile

Lezione 10. Parte 1: coppie generiche *(50 minuti).* Molte costruzioni in Java vengono ripetute uguali per tipi diversi. Per ogni tipo T, S (con T,S = interi, booleani, reali, stringhe o classi qualsiasi) possiamo definire una classe i cui oggetti sono coppie di un oggetto di tipo T e un oggetto di tipo S. Possiamo definire la classe dei nodi il cui contenuto ha tipo T, per un qualunque tipo T. Possiamo definire la classe delle pile e delle code di oggetti di tipo T. Per non ripetere la costruzione per ogni tipo T possiamo introdurre variabili di classe e costruire una sola volta la classe delle coppie di tipo T, S o la classe dei nodi/pile di tipo T, e poi scegliere i tipi T, S.

Una classe è detta generica se contiene nella sua definizione variabili di classe. Vediamo una definizione della classe ***GenericPair*** delle coppie di tipo T, S come classe generica con variabili di tipo T,S.

Per inserire le variabili di classe usiamo delle parentesi angolose: ***GenericPair<T,S>.*** Per definire una classe generica di coppie prendo la definizione di una qualunque classe C di coppie concrete di tipi T0, S0 (per es., T0=int e S0=double), e rimpiazzo C con GenericPair<T,S>, e T0, S0 con T,S. Fa solo ***eccezione il costruttore***, che dobbiamo chiamare *GenericPair* quando lo definiamo, e *GenericPair<T,S>* quando lo usiamo. Devo anche cancellare tutti i metodi che contengono riferimenti a caratteristiche particolari di T0, S0: il codice che resta deve poter funzionare con due classi ***qualunque*** T, S. In una classe generica posso utilizzare metodi che esistono in tutte le classi, per esempio: ***boolean equals(T x), String toString()***.

```
//GenericPair.java
public class GenericPair<T,S> {
 private T first; private S second;

/* Dobbiamo chiamare il costruttore GenericPair, senza <T,S>*/
 public GenericPair(T first, S second)
 {this.first = first; this.second = second;}

 public T getFirst() { return first;  }
```

```
 public S getSecond(){ return second; }

 public void setFirst(T first)  { this.first=first;   }
 public void setSecond(S second){ this.second=second; }

// Il metodo toString() appartiene a qualunque classi T, U
// quindi puo' essere usato

 public String toString()
{return "(" + first.toString() + "," + second.toString() + ")";}}
```

Come esempio di metodo ***statico generico*** definiamo un metodo ***GenericPair<S,T>***, che dati i tipi S,T e una coppia "inverte" le componenti della coppia. Il metodo costruisce una nuova coppia e non altera quella originaria. S e T sono parametri di tipo del metodo e vanno indicati prima del tipo di ritorno. Se un metodo pubblico statico prende in input tipi T, S e una lista (…) di dati, e ha tipo di ritorno la classe generica C<T,S>, dobbiamo dichiararlo così:

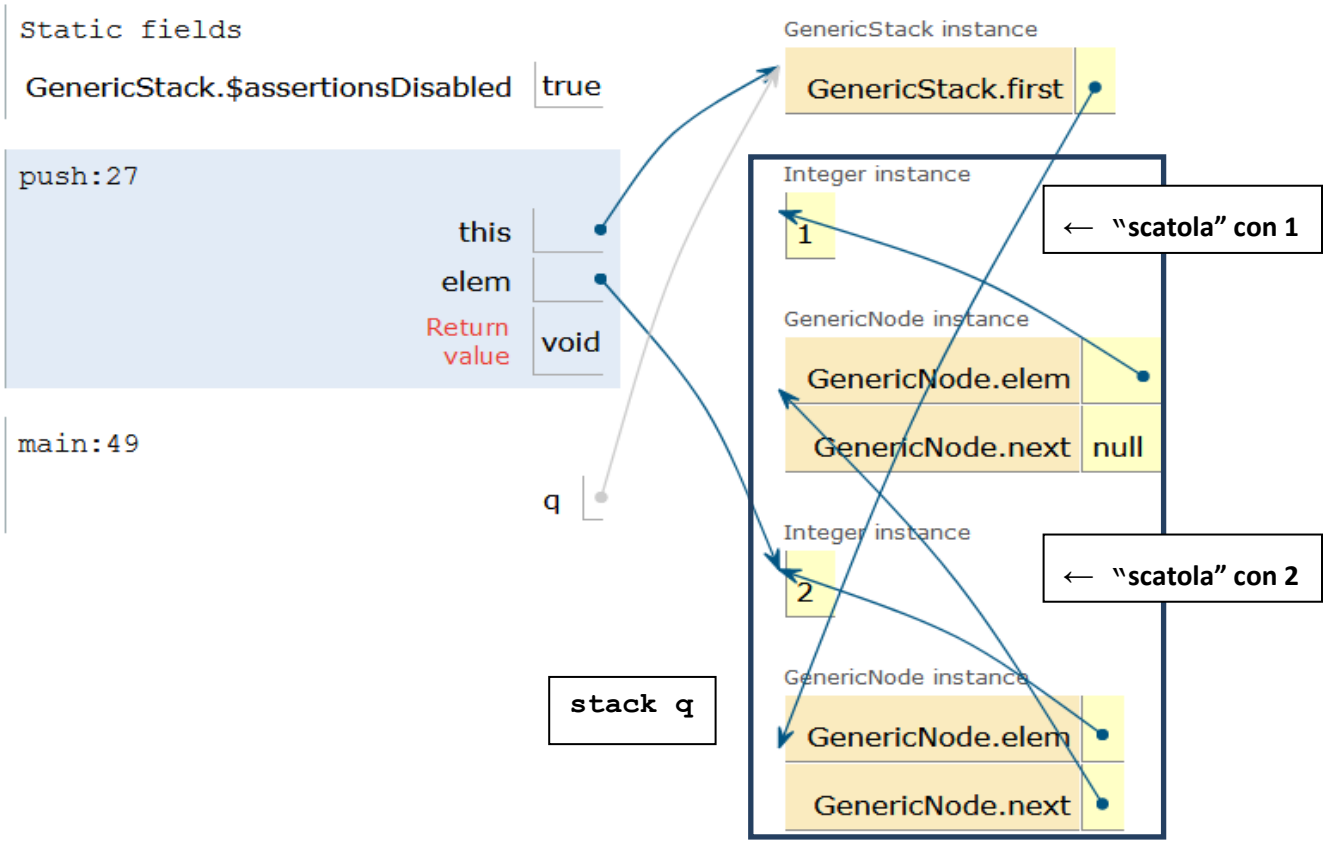
```
public static <T,S>  C<T,S> metodo(…)
```

```
//TestGenericPair.java
public class TestGenericPair {

public static <T,S> GenericPair<S,T> inv(GenericPair<T,S> p)
{return new GenericPair<S,T>  (p.getSecond(), p.getFirst());}

 public static void main(String[] args)
{ // nel seguito, "\t" e' la tabulazione e inserisce spazi bianchi
 GenericPair<String,Integer> p = new GenericPair<>("pluto", 1);
 System.out.println( "p = \t\t\t\t"     + p);
 System.out.println( "inv(p) = \t\t\t"  + inv(p));
 System.out.println( "p non cambia: \t" + p);
 //Dato che inv e' un metodo statico, al di fuori della sua classe
 //deve venire chiamato come: TestGenericPair.<String,Integer>inv(p);
  }}
```

Lezione 10. Parte 2: nodi generici e tipo Integer *(50 minuti).* Definiamo la classe ***GenericNode*** dei nodi con elementi di un tipo qualsiasi T. GenericNode si ottiene indicando un parametro di tipo T tra parentesi angolari dopo il nome GenericNode della classe. All'interno della classe il tipo T può essere usato (quasi) ovunque deve esserci un

tipo, dunque nelle dichiarazioni di attributi, parametri di metodi, e variabili locali. La definizione di GenericNode<T> è ottenuta modificando la definizione della classe Node (liste di interi) della Lezione 08, rimpiazzando int con la generica classe T, e Node con GenericNode<T>.

Autoboxing: i tipi Integer, Boolean e Double. Una difficoltà nell'uso dei generici è che Java considera una classe come un insieme di indirizzi di dati: in base a questa definizione i tipi primitivi int, bool, double ***non sono classi*** e non possono quindi essere sostituiti alla variabile di classe T. Per ovviare, Java ci fornisce le classi Integer, Boolean, Double, detti tipi ***wrapper***, fatte da un indirizzo di un intero, booleano e reale. Immaginiamo queste classi fatte di ***boxes ("scatole")*** che contengono un intero, booleano, reale, anziché fatte di interi, booleani e reali. int e Integer sono due versioni ***equivalenti*** degli interi, possiamo scrivere uno al posto dell'altro senza problemi, se necessario Java trasforma un int in un Integer e viceversa con operazioni dette ***autoboxing ("inscatolamento")***. Lo stesso vale per bool, Boolean, double, Double.



Nell'immagine rappresenta li è una pila dinemica con i controlli tramite asserzioni

Qui sopra vediamo come esempio una lista q={1,2} i cui elementi hanno tipo Integer anziché int. La lista q contiene ***gli indirizzi di 1,2*** (chiamati *"integer instances"* e non interi), disegnati come una ***"scatola"*** attorno a 1,2. L'immagine si riferisce all'istante in cui inseriamo 2 eseguendo q.push(2).

```
// GenericNode.java
public class GenericNode<T> {
 private T elem;
 private GenericNode<T> next;

 public GenericNode(T elem, GenericNode<T> next)
 {this.elem = elem; this.next = next;}

 public T getElem(){return elem;}
 public GenericNode<T> getNext(){return next;}

 public void setElem(){this.elem=elem;}
 public void setNext(GenericNode<T> next)
  {this.next=next;}}
```

Definiamo una classe ***GenericStack<T>*** generica per rappresentare stack di elementi di tipo T, con T arbitrario.

```
import java.util.*;

public class GenericStack<T> {
    private GenericNode<T> first;

public GenericStack(){first = null;}

public boolean empty(){ return first == null; }

public void push(T elem){first = new GenericNode<>(elem, first);}

public T pop(){assert !empty(): "pop on empty stack";
 T x = first.getElem();
 first = first.getNext();
 return x;}

public void scriviOutput()
```

```
{ GenericNode<T> s = first;
  while(s!=null)
  {System.out.println(s.getElem());
   s=s.getNext();}}
}
```

Sperimentiamo la classe ***GenericStack<T>*** generica per costruire pile di tipi diversi: p pila di String, q pila di Integer, s pila di Double.

```
import java.util.*;

public class TestGenericStack {

public static void main(String[] args){
// Creiamo uno stack per contenere stringhe
System.out.println( "Stampo p = {\"hello \", \"world!\"} ");
 GenericStack<String> p = new GenericStack<>(); //p pila String
 p.push("hello ");
// OK: il metodo push si aspetta un argomento di tipo String
 p.push("world!"); p.scriviOutput(); // stampo 2 strighe
 String s1 = p.pop();
// OK: il metodo pop ritorna un valore di tipo String
 String s2 = p.pop();
 p.push(s2 + s1); // OK: s2 + s1 produce una nuova stringa

// p.push(1);
// ERRORE: non posso inserire int in uno stack di String

// Creiamo uno stack per contenere numeri interi. Notiamo che
// NON e` possibile usare tipi primitivi int, boolean double
// per istanziare classi generiche, dunque DOBBIAMO usare il tipo
// Integer (i numeri devono comparire "inscatolati" nello stack)
 System.out.println( "Stampo q = {1,2} ");
 GenericStack<Integer> q = new GenericStack<>(); //q pila Integer
 q.push(1); /* OK: il metodo push si aspetta un argomento di tipo
Integer, gli forniamo un int che puo' essere convertito in Integer
grazie all'autoboxing */
 q.push(2); q.scriviOutput();// stampo 2 interi
```

```
 q.push(q.pop() + q.pop()); /* OK: il metodo pop ritorna un Integer
da cui Java estrae automaticamente un int nel momento in cui vede che
usiamo il valore per un'operazione primitiva (+) */

 // q.push("hello"); // ERRORE: non posso inserire String in
 // uno stack di Integer

 // Inserisco alcuni numeri casuali tra 0 e 1 in una pila s di Double
 Random r = new Random();//r = generatore numeri casuali
 GenericStack<Double> s = new GenericStack<Double>(); //s pila Double
 //Scelgo a caso la dimensione dello stack, al massimo 20 elementi
 int n = r.nextInt(20);
 //Scelgo a caso ogni elemento dello stack e lo aggiungo a s
 for (int i = 0; i < n; i++) s.push(r.nextDouble());

/* Possiamo usare il metodo printStack per stampare il contenuto di
Stack di elementi di tipo arbitrario (ma tutti dello stesso tipo)*/
 System.out.println( "--->(1) ora p e' uno stack di 1 stringa");
 p.scriviOutput(); // OK: p e' uno Stack di String
 System.out.println( "--->(2) ora q e' uno stack di 1 Integer");
 q.scriviOutput(); // OK: q e' uno Stack di Integer
 System.out.println( "--->(3) s e' uno stack di " + n + " Double");
 s.scriviOutput(); // OK: s e' uno Stack di Double
}}
```

Lezione 11 Ereditarietà e assert

Lezione 11. Parte 1. Un primo esempio di estensione di una classe: la classe BottigliaConTappo.

Vediamo come definire una nuova classe D da una classe data C, aggiungendo nuovi attributi/metodi e riscrivendo una parte dei metodi già esistenti. D viene detta "estensione" o "sottoclasse" di C. Come esempio, riprendiamo la classe Bottiglia della Lezione 05 e aggiungiamo alla bottiglia due stati, aperto/chiuso, con la regola che una bottiglia per dare o ricevere acqua deve essere aperta. Di conseguenza, alcuni metodi devono venir modificati. Chiamiamo la classe così ottenuta ***BottigliaConTappo***.

Per cominciare, rivediamo rapidamente la definizione della classe ***Bottiglia***. Una bottiglia ha una capacità (non modificabile) e un livello (modificabile). Oltre ai metodi get, set aggiungiamo metodi per aggiungere e rimuovere una quantità a una bottiglia (nei limiti del possibile). Per evitare modifiche alla capacità non forniamo un metodo get per la capacità.

```
// Bottiglia.java
public class Bottiglia
{ // quantita' intere espresse in litri
 private int capacita; // 0 <= capacita
 private int livello;  // 0 <= livello <= capacita

 public Bottiglia(int capacita)
 {this.capacita = capacita;
  livello = 0;
  assert (0<=livello) && (livello <= capacita);}

/* aggiungiamo tutta la parte di una quantita' data che trova posto
nella bottiglia e restituiamo la quantita effettivamente aggiunta */
 public int aggiungi(int quantita)
 {assert quantita >= 0;
  int aggiunta = Math.min(quantita, capacita-livello);
  livello = livello + aggiunta;
  assert (0<=livello) && (livello <= capacita);
  return aggiunta;}

/* Rimuoviamo la quantita' richiesta se c'e', altrimenti togliamo
tutto, restituiamo la quantita' effettivamente rimossa */
```

```
  public int rimuovi(int quantita)
  {int rimossa = Math.min(quantita, livello);
   livello = livello - rimossa;
   assert (0<=livello) && (livello <= capacita);
   return rimossa;}

  public int getCapacita(){ return this.capacita; }
  public int getLivello() { return this.livello; }
  public void setLivello(int livello)
  {this.livello = livello;
   assert (0<=livello) && (livello <= capacita);}

  public void scriviOutput()
  {System.out.println(" " + livello + "/" + capacita);}}
```

Estensione di una classe. Aggiungiamo un attributo alla classe Bottiglia dotandola di un tappo che può essere in due stati (aperto o chiuso) e facendo in modo che il versamento di liquido dalla e nella bottiglia abbia effetto solo quando la bottiglia è aperta. Questa operazione viene chiamata "estensione" di una classe.

Estensioni come sottoclassi. Ci conviene immaginare una classe D estensione di una classe C come una ***sottoclasse di C***. Nel nostro esempio, alcuni oggetti della classe C = Bottiglia hanno un tappo e due stati, e si trovano nella sottoclasse D = BottigliaConTappo. Altri oggetti non hanno il tappo e non si trovano nella classe ma non nella sottoclasse.

L'estensione consente di: aggiungere attributi/metodi, riutilizzare attributi/metodi pubblici, rendere pubblico un attributo/metodo privato (ma senza poter utilizzare la versione privata), riscrivere ***("override")*** il corpo di un metodo, estendere la classe in cui questo metodo restituisce il risultato. Nel nostro caso, dobbiamo fare override dei metodi "aggiungi" e "rimuovi", per tener conto che queste azioni richiedono una bottiglia aperta.

Sintassi per la classe estesa. Scriviamo "class D extends C" per definire una estensione D di C. All'interno della definizione di D, con "super" ("sovraclasse") indichiamo la classe D che estendiamo, con "super(...)" un costruttore di D, con "super.metodo(...)" un metodo di D.

```
// BottigliaConTappo.java
public class BottigliaConTappo extends Bottiglia {
```

```
/* NUOVO attributo privato per memorizzare lo stato della bottiglia
(true = bottiglia aperta, false = bottiglia chiusa) */
 private boolean aperta;

/* NUOVO costruttore. Spesso dobbiamo definire un costruttore per le
classi estese: raramente il costruttore di default fornito da Java
per una estensione e' sensato */
 public BottigliaConTappo(int capacita){
 /* invochiamo il costruttore della classe base per fare le
inizializzazioni della capacita' */
 super(capacita);
 // supponiamo che la bottiglia sia inizialmente chiusa
 aperta = false;}

// NUOVO metodo get per sapere se la bottiglia e` aperta o chiusa
 public boolean aperta(){ return aperta; }

// NUOVO metodo per aprire la bottiglia
 public void apri()     { aperta = true; }

// NUOVO metodo per chiudere la bottiglia
 public void chiudi()   { aperta = false; }

// Ereditiamo i metodi get, set e scriviOutput() da Bottiglia

/* OVERRIDE del metodo "aggiungi" per versare liquido nella
bottiglia: richiediamo che la bottiglia sia aperta. Dal momento che
"aggiungi" deve restituire la quantita` di liquido aggiunto anche nel
caso in cui la bottiglia sia chiusa, dobbiamo restituire un valore
sensato (0 in questo caso) */
public int aggiungi(int quantita){
 if (aperta)
  return super.aggiungi(quantita);
 else
   return 0;}

/* OVERRIDE del metodo "rimuovi" per versare liquido dalla bottiglia:
richiediamo che la bottiglia sia aperta. Dal momento che "rimuovi"
deve restituire la quantita' di liquido tolto anche nel caso in cui
la bottiglia sia chiusa, dobbiamo restituire un valore sensato (0 in
questo caso) */
 public int rimuovi(int quantita)
```

```
if (aperta)
  return super.rimuovi(quantita);
 else
   return 0;}

/* Controlliamo che lo stato "bottiglia aperta" lascia invariati i
travasi, e che lo stato "bottiglia chiusa" li azzera. */
// BottigliaConTappoDemo.java
public class BottigliaConTappoDemo {
 public static void main(String[] args){
 System.out.println( "Definisco b da 100 litri vuota e la apro");
 BottigliaConTappo b = new BottigliaConTappo(100); b.apri();
 b.scriviOutput();
 System.out.println( " b.aperta() = "     + b.aperta());

 System.out.println( "Aggiungo 50 litri in b poi chiudo b");
 System.out.println( " b.aggiungi(50) = " + b.aggiungi(50));
 b.chiudi();
 System.out.println( " b.aperta() = "     + b.aperta());

 System.out.println( "Chiedo di rimuovere 20 litri da b: zero");
 System.out.println( " b.rimuovi(20) = "  + b.rimuovi(20));
 System.out.println( " b.getLivello() = " + b.getLivello());

 System.out.println( "Apro b: ora riesco a togliere 20 litri");
 b.apri();
 System.out.println( " b.aperta() = "     + b.aperta());
 System.out.println( " b.rimuovi(20) = "  + b.rimuovi(20));
 System.out.println( " b.getLivello() = " + b.getLivello());}}
```

Lezione 11. Parte 2. "Assert o non assert, questo è il problema!" *(di Luca Padovani)*.

Ripassiamo l'uso dell'assert in Java e vediamo un esercizio di un esame che lo riguarda.

assert e **if** hanno funzioni diverse. Vediamo quando usare l'uno e l'altro.

assert indica una condizione (pre-condizione, post-condizione, invariante) che il programmatore ritiene vera in una determinata riga del programma, e che vuole controllare. In particolare, **assert** non modifica in alcun modo l'esecuzione del programma, salvo farlo terminare nel momento in cui la condizione attesa non è vera.

if serve invece a controllare il usso di esecuzione del programma a seconda di una condizione che potrebbe legittimamente essere sia vera che falsa. Vediamo un esempio tipico di utilizzo di ciascun costrutto:

```
public static int abs(int x) {if (x >= 0) return x; else return -x;}
```

Qui il programmatore ha definito la funzione "valore assoluto". Il parametro x può legittimamente essere positivo oppure negativo ed il comportamento della funzione varia a seconda dei due casi. Di conseguenza, il costrutto giusto è l'if.

```
public static int sqrt(int x)  {assert (x >= 0);
  return (int) Math.sqrt((double) x);}
```

Qui il programmatore ha definito la funzione "radice quadrata" per numeri interi in termini dell'analoga funzione per numeri **double**. La funzione è definita solo per un sottoinsieme di tutti i possibili valori del parametro x, in particolare per quelli non negativi. Questa restrizione è documentata da un'asserzione. Nel caso la condizione risulti essere non verificata, la responsabilità è da attribuire ad un'altra regione del programma (presumibilmente a chi applica la funzione), non a sqrt.

Non sempre vi è una distinzione così netta tra i casi in cui è appropriato usare if e quelli in cui è appropriato usare assert. Prendiamo ad esempio il metodo push di un'ipotetica classe Stack che implementa una pila di capacità finita. Vi sono almeno due modi di definire tale metodo.

```
public void push(int x){assert (size<data.length);data[size++] = x;}
```

Qui il programmatore assume che l'utilizzatore della pila si assicuri che la pila non sia piena prima di effettuare la push. Se tale condizione risulta falsa, la colpa è da attribuire all'utilizzatore.

```
public boolean push(int x) {if (size < data.length) {data[size] =
x;++size;return true;} else return false;}
```

Qui il programmatore ha definito una versione "robusta" di push che verifica con un if se l'operazione è possibile e notifica l' utilizzatore dell'esito dell'operazione con un valore booleano. Sta poi all'utilizzatore gestire (eventualmente) il caso in cui l'operazione sia fallita. Entrambe le implementazioni di push possono essere ragionevoli a seconda del contesto. Ad esempio, se la classe Stack non fornisce alcun metodo pubblico per verificare se una pila è piena, è evidente che la prima implementazione di push fa

un'assunzione discutibile. D'altro canto, lasciare all'utilizzatore l'onere di gestire il caso di una push con pila piena (seconda implementazione) è rischioso. Se l'utilizzatore invoca push dimenticandosi poi di controllare se l'operazione ha avuto successo, il programma continua l'esecuzione in uno stato in cui alcuni valori non sono affidabili, senza segnalare l'errore. Abbiamo il ***vantaggio*** che il programma non si spegne completamente (pensate a un programma che gestisce un aereo), ma lo ***svantaggio*** che ritardiamo il momento in cui ci rendiamo conto dell'errore.

Occorre infine sottolineare uno svantaggio generale nella segnalazione di un errore/problema attraverso il valore di ritorno di un metodo. Questa tecnica costringe il programmatore a sacrificare uno o più valori di ritorno che diventano "segnali di errore". Nel caso della seconda implementazione di push qui sopra il sacrificio è tollerabile in quanto il metodo non restituisce alcun risultato (come nella prima implementazione). In generale però può non essere facile individuare un valore da usare come "segnale di errore" e tale valore non è necessariamente descrittivo del tipo di problema che è avvenuto. Un esempio concreto di questo problema si può osservare nella seguente versione (apparentemente "robusta") di pop:

```
public int pop(){if (size > 0) return data[--size]; else return -1;}
```

Qui il programmatore della classe Stack ha scelto di usare il numero -1 come segnale del fatto che la pila è vuota e dunque non vi è un elemento da estrarre con pop. Purtroppo, dal punto di vista dell'utilizzatore della classe non vi è alcun modo per distinguere il caso in cui la pila è vuota (e pop ritorna -1 per segnalare il problema) dal caso in cui la pila contiene almeno un elemento e quello in cima è proprio -1.

Il meccanismo migliore finora trovato per gestire situazioni inattese (come un tentativo di push con pila piena o un tentativo di pop con pila vuota) è quello delle ***eccezioni***, che sarà descritto in una parte più avanzata del corso. Tale meccanismo consente al programmatore di:

- ***separare nettamente*** le segnalazioni di errori dalla restituzione "normale" di valori, senza alcun sacrificio per questi ultimi;
- definire ***messaggi di errori*** che possono contenere dettagli sul problema che si è verificato;
- lasciare all'utilizzatore di un metodo la scelta se gestire il problema ***(catturando l'eccezione, come vedremo)*** o lasciar terminare il programma.

Ora vediamo un esempio di esercizio di esame che riguarda un assert.

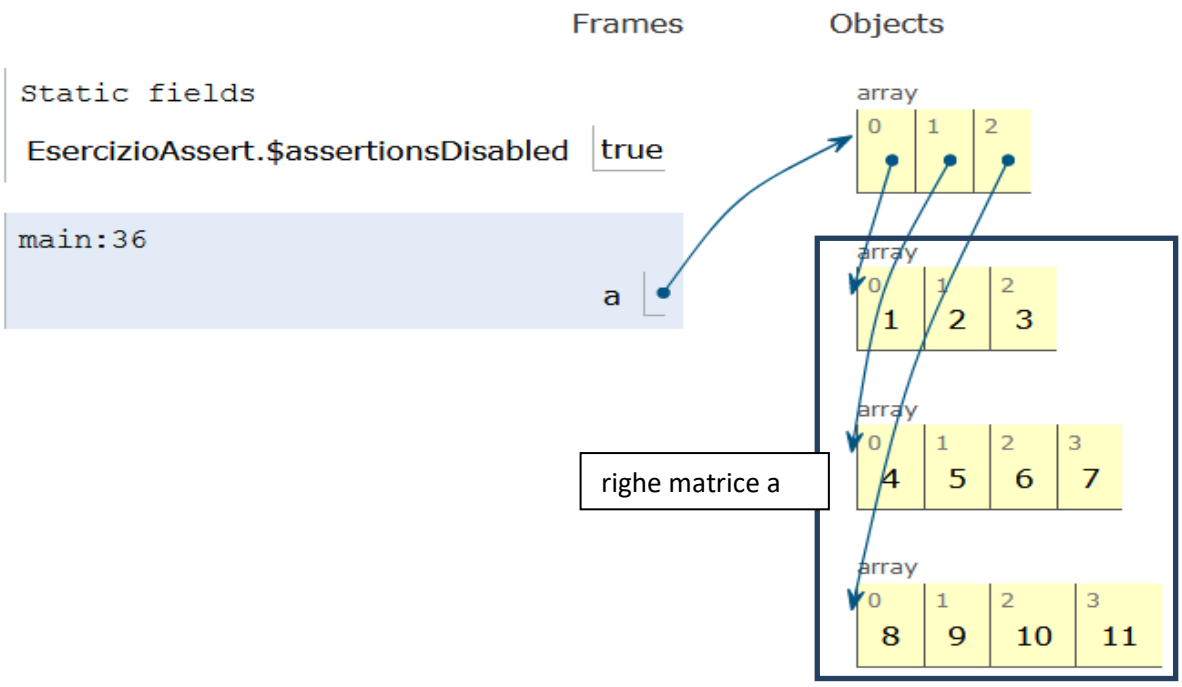
Esame di ProgII del 2017-06-14. Esercizio 3 **(6 punti).**

Sia dato il metodo

```
public static boolean metodo(int[][] a)
{boolean ris=true;
 for (int i = 0; i < a.length; i++)
  for (int j = 0; j < a.length; j++)
   if (a[i][j] != a[j][i]) ris=false;
 return ris;}
```

1. Determinare sotto quali condizioni il metodo viene eseguito correttamente (cioè senza lanciare alcuna eccezione) e scrivere una corrispondente asserzione da aggiungere come pre-condizione per il metodo. Nello scrivere l'asserzione è possibile fare uso di eventuali ***metodi statici ausiliari*** che vanno comunque definiti anche se visti a lezione.
2. Descrivere in modo conciso e chiaro, in non più di 2 righe di testo, l'effetto del metodo.

Un esempio di vettore di vettori:
a={{1,2,3},{4,5,6,7},{8,9,10,11}}



Nell'immagine vi è rappresentata un vettore di vettori tramite array

Definiremo l'asserzione per il metodo usando un metodo ausiliario:

public static boolean ok(int[][] a){…}

Soluzione dell'esercizio 3 di esame (Lezione 11)

```
//EsercizioAssert.java
public class EsercizioAssert
{
 public static boolean metodo(int[][] a)
 {assert ok(a): "metodo(a) solleva eccezioni";
//ok(a)=true se e solo se: metodo(a) non solleva eccezioni
  boolean ris=true;
  for (int i = 0; i < a.length; i++)
   for (int j = 0; j < a.length; j++)
    if (a[i][j] != a[j][i]) ris=false;
  return ris;}

/* a = vettore di vettori-riga di interi. Per esempio:
```

a[0][0]	a[0][1]	a[0][2]	
a[1][0]	a[1][1]		
a[2][0]	a[2][1]	a[2][2]	a[2][3]

```
(1) metodo(a) solleva un'eccezione se e solo se a=null o se per
qualche riga a[i], o a[i]=null oppure a[i] ha lunghezza inferiore a r
= numero righe a. (2) Altrimenti metodo(a)= true se e solo se
la matrice rxr nel lato sinistro di a e' simmetrica. */


//Definiamo il metodo ausiliario ok(a) usato per l'assert
public static boolean ok(int[][] a)
 {if (a==null) return false;
//se a=null allora a.length produce NullPointerException

  int r = a.length; int i=0;
  while(i<r){if ((a[i]==null)||(a[i].length<r) return false; ++i;}
//se a[i]=null allora a[i].length produce NullPointerException
//se a[i]<r allora a[i][r-1] produce ArrayOutOfBoundException

  return true;} //se nulla di cui sopra capita: ok(a)=true

 public static void main(String[] args)
{int[][] a=new int[3][];
// a = { null, null, null}
 a[0]=new int[3]; a[1]=new int[4]; a[2]=new int[4];
```

```
// a = { {0,0,0}, {0,0,0,0}, {0,0,0,0}}
 a[0][0]=1; a[0][1]=2; a[0][2]=3; a[1][0]=4; a[1][1]=5; a[1][2]=6;
 a[1][3]=7; a[2][0]=8; a[2][1]=9; a[2][2]=10; a[2][3]=11;
// a = { {1,2,3}, {4,5,6,7}, {8,9,10,11}}
 System.out.println( "metodo(a)=" + metodo(a));
// ok(a)=true e metodo(a)=false perche':
// la matrice quadrata 3x3 nel lato sinistro di a non e' simmetrica

 int[][] b=new int[3][]; b[0]=new int[3]; b[1]=new int[2];
 b[2]=new int[4]; //b = {{0,0,0}, {0,0}, {0,0,0,0}}
// ok(b)=false e metodo(b) solleva una eccezione se eseguiamo:
// System.out.println( "metodo(b)=" + metodo(b));
// perche': la seconda riga di b ha meno di 3 elementi
}}
```

Lezione 12 Estensioni ripetute di classi

Lezione 12. Un esempio di estensioni ripetute di classi. Partiamo dalla classe ***DynamicStack*** delle pile dinamiche (Lezione 08), e aggiungiamo prima un attributo ***"max"*** (massimo valore), poi un attributo ***"size"*** (numero degli elementi). Nella classe DynamicStack.java sostituiamo "private" con **"protected"** davanti a "top". Questa keyword consente di utilizzare "top" in ogni classe che estende la classe data, ma non nelle altre classi. Estendere comporta aggiungere all'invariante di classe delle condizioni su max e size.

```
// classi Node.java e DynamicStack.java: copiatele dalla Lezione 08
// Dopo aver copiato, nella definizione di DynamicStack sostituite:
//        "private Node top"       con       "protected Node top"
// Otterrete:

public class DynamicStack{
protected Node top;
public DynamicStack(){top = null;}
public boolean empty(){return top==null;}
public void push(int x) {top = new Node(x,top);}
public int pop(){assert !empty();int x = top.getElem();
   top = top.getNext(); return x;}
public int top(){assert !empty(); int x = top.getElem();return x;}
public void scriviOutput()
{Node temp = top;
  while (temp != null) {System.out.println( " || " + temp.getElem());
      temp=temp.getNext();}}
public DynamicStack(int n)
{top = null; int i = 1; while (i<=n) {top = new Node(i,top);i++;}}}


//DynamicStackMax.java
public class DynamicStackMax extends DynamicStack {
// Manteniamo il costruttore top di tipo Node e aggiungiamo
private int max;
/* INVARIANTE di classe di DynamicStack: top punta alla cima della
pila. Aggiungiamo: SE lo stack non e` vuoto, allora max contiene il
massimo valore dello stack, se lo stack e' vuoto il valore di max e'
arbitrario */
```

```
/* COSTRUTTORE. Dobbiamo spesso fornire un costruttore per le classi
estese: raramente il costruttore di default fornito da Java per una
estensione e' sensato */

public DynamicStackMax(){super();
//Invoco il costruttore della classe superiore con 0 argomenti
  max = 0;
// inizializziamo il nuovo attributo, max, anche se il suo valore non
// ha senso quando lo stack e` vuoto. Quando lo stack e' vuoto non
// consentiremo l'uso di max.
}

// NUOVO metodo get per il nuovo campo max
public int getMax()
{assert !empty(); // se pila vuota: non corretto chiedere il massimo
  return max;}

// OVERRIDE del metodo push(int n): inseriamo di un elemento in cima
// alla pila aggiornando il valore del massimo
public void push(int n)
{if (empty())
   max=n;
//se la pila e' vuota il massimo e' l'elemento n appena inserito
  else
//altrimenti e' il massimo tra elemento inserito e il max. precedente
    max = Math.max(max, n);
  super.push(n); //invoco il push della classe superiore
}

// NUOVO metodo per ricalcolare max, se abbiamo motivo per
// dubitare che max sia davvero il massimo della pila

// Nota: possiamo usare il nodo "top" della pila perche' abbiamo
// dichiarato top "protected" e quindi accessibile nelle classi che
// estendono DynamicStack

private void resetMax()
 {if (!empty()) //se la pila e' vuota ogni valore di max va bene
  // altrimenti ricalcolo il massimo della pila
  {max = top.getElem();
// calcolo il max. tra il primo elemento della pila e gli altri
// per evitare di modificare l'indirizzo top della pila introduco
```

```
// una nuova variabile p di tipo nodo con valore iniziale top
   for (Node p = top.getNext(); p != null; p = p.getNext())
         max = Math.max(max, p.getElem());}
}

// OVERRIDE di pop(): rimozione di un elemento dalla cima della pila
// Attenzione: puo' richiedere il ricalcolo del massimo
public int pop()
{
 assert !empty();
 int n = super.pop(); //invoco il pop() della classe superiore
//Se l'elemento tolto e' il massimo allora il massimo puo' cambiare
// e quindi va ricalcolato
 if (n == max) resetMax();
 return n;}

//EREDITA' Il metodo top() e' ereditato, non deve essere riscritto:
//leggere l'elemento in cima alla pila non cambia il max della pila

//OVERRIDE del metodo di scrittura
public void scriviOutput()
 {super.scriviOutput(); System.out.println( " || max= " + max);}}
```

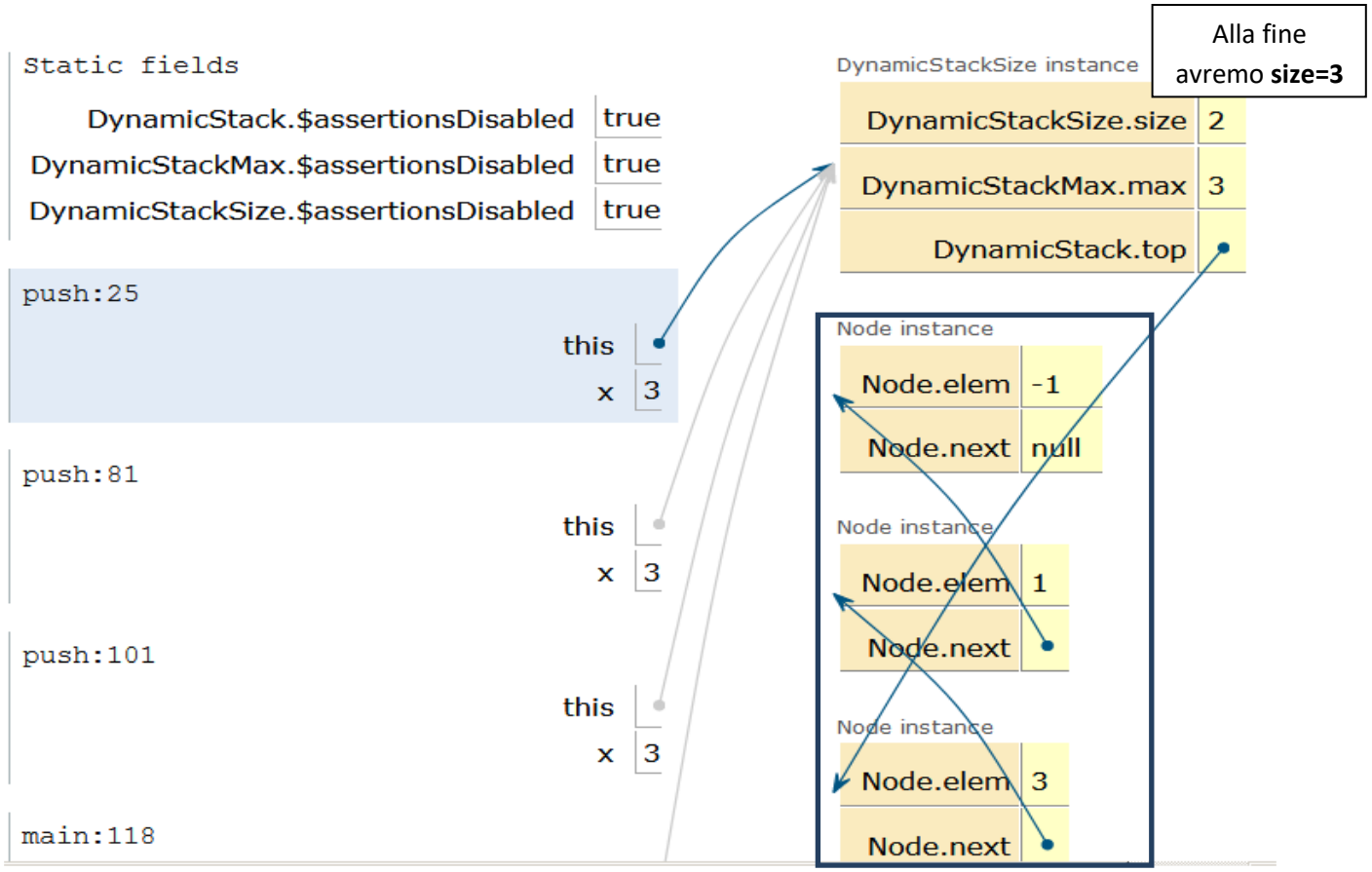
Ora estendiamo la classe estesa ***DynamicStackMax*** aggiungendo un attributo con il conto degli elementi della pila. Chiamiamo il risultato ***DynamicStackSize***.

```
//DynamicStackSize.java
public class DynamicStackSize extends DynamicStackMax
{private int size; // Aggiunta all'INVARIANTE di classe:
// "size" = numero elementi sullo stack

//COSTRUTTORE Dobbiamo quasi sempre definire un costruttore per le
// estensioni il costruttore di default in genere non e' affidabile
 public DynamicStackSize()
 {super(); //Invoco il costruttore della classe superiore:0 argomenti
  size = 0;}

// NUOVO metodo get per il nuovo campo size
public int getSize() { return size; }
```

```
// OVERRIDE del metodo push: inserimento elemento in cima alla pila
public void push(int n)
{super.push(n); //invoco il metodo push(n) della classe superiore
 size++;        //aggiorno il numero degli elementi
}

// OVERRIDE del metodo pop: rimozione elemento dalla cima della pila
public int pop(){assert !empty();
 size--;             //aggiorno il numero degli elementi
 return super.pop(); //invoco il metodo pop() della classe superiore
}

//EREDITA' top() viene ereditato e non deve essere riscritto: leggere
//l'elemento in cima alla pila non cambia il size della pila

//OVERRIDE del metodo di scrittura
public void scriviOutput()
 {super.scriviOutput(); System.out.println( " || size = " + size);}}


// Sperimento la classe DynamicStackSize
//DynamicStackSizeDemo.java
public class DynamicStackSizeDemo
{public static void main(String[] args)
{System.out.println( "Definisco la pila P = {-1}");
 DynamicStackSize P = new DynamicStackSize();
 P.push(-1); P.scriviOutput();

 System.out.println( "Definisco la pila P = {-1,1,3,5,7,9,11}");
 P.push(1); P.push(3); P.push(5); P.push(7); P.push(9);
 P.push(11); P.scriviOutput();

 System.out.println( "Estraggo 11, 9, 7. Leggo 5");
 System.out.println( " P.pop() = " + P.pop());
 System.out.println( " P.pop() = " + P.pop());
 System.out.println( " P.pop() = " + P.pop());
//Leggiamo il prossimo elemento, 5, senza estrarlo dala pila
 System.out.println( " P.top() = " + P.top());
 System.out.println( "Stampo cio' che resta: P={-1,1,3,5}");
 P.scriviOutput(); }}
```

Un esempio di calcolo su una pila P di DiynamicStackSize

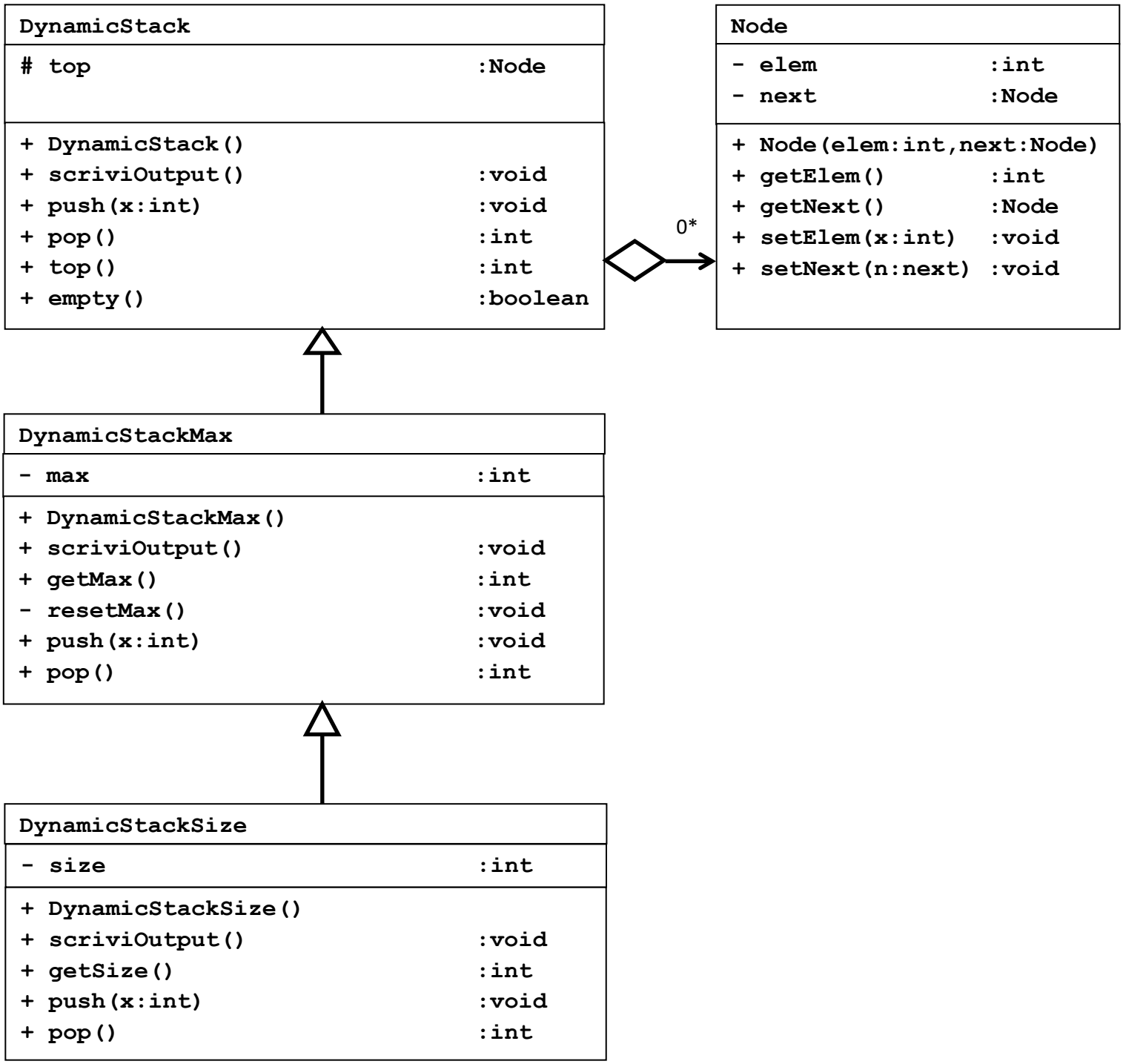
Partiamo da ***P={-1}*** e eseguiamo ***P.push(3)***. Ecco cosa succede. La pila ha tre attributi, l'indirizzo dell'elemento in cima alla pila che fa parte della classe ***DynamicStack***, e i due attributi ***max*** e ***size*** aggiunti con le due estensioni. Il metodo push(3) della classe ***DynamicStackSize*** richiamo il metodo push(3) della classe ***DynamicStackMax***, che aggiorna max e richiama il metodo push(3) della classe ***DynamicStack***. Alla fine il primo push(3) aggiornerà *size*.



Nella classe DynamicStackSize, P.push(3) richiama altri due P.push(3)

Diagramma UML per pile dinamiche e le loro estensioni

Una pila dinamica è definita ***aggregando*** 0 o più elementi della classe Node. Partendo da quest osservazione definiamo il diagramma UML per DynamickStack e Node. Le estensioni ***DynamickStackMax*** e ***DynamickStackSize*** si indicano con una ***freccia verso l'alto*** dalla classe che estende verso la super-classe. In un diagramma UML, un attributo ***protected*** ("top" nel nostro caso) si indica con ***#***.



Nelle tabelle vi è la rappresentazione delle catene push e pop delle DynamicStack

Lezione 13 Tipo esatto e binding dinamico

Lezione 13. Parte 1. Tipo esatto e tipo apparente. Un esempio di Downcast.

Supponiamo che C sia una classe Java che estende una classe D. Allora ogni oggetto di tipo C viene considerato anche un oggetto di tipo D. Questo è possibile perché ogni oggetto di tipo C ha tutti gli attributi di un oggetto di tipo D, più possibilmente degli attributi specifici di C. Basta ignorare gli attributi specifici di C (senza per questo cancellarli), per ottenere un oggetto di tipo D. Questa operazione si chiama ***Upcast***. Grazie all'Upcasting, un oggetto in Java può avere diversi tipi. Chiamiamo ***"tipo esatto"*** di un oggetto il tipo con cui l'oggetto è stato costruito: allora l'oggetto può avere come tipo tutti e soli i tipi che includono il suo tipo esatto.

L'operazione opposta di ***Upcast*** si chiama ***Downcast***. Per esempio, se obj ha tipo D, e C è inclusa in D, allora ***((C) obj)*** indica "obj con tipo C" se il tipo esatto di obj è C oppure incluso in C. Altrimenti calcolare ((C) obj) solleva una eccezione (detta ***ClassCastException***) e termina il programma.

In ogni riga di programma Java si chiama ***"tipo apparente"*** di una espressione il tipo di questa espressione in un comando Java. Il primo controllo fatto da Java è che il tipo apparente di una espressione sia quello richiesto dal comando che la usa, oppure sia incluso in esso. Se non è così il programma **non compila**.

Se il programma compila, durante l'esecuzione Java utilizza il tipo esatto di un oggetto obj per ***decidere quale versione di un metodo applicare all'oggetto***. Questo meccanismo viene chiamato ***binding dinamico.*** Se Java deve calcolare quando deve calcolare obj.m(…) e ci sono diverse versioni sovrascritte di m(…), Java parte dal tipo esatto C di obj, e cerca il metodo m nella classe C. Se non lo trova Java cerca il metodo m nella classe D di cui C è estensione, nella classe E di cui D è estensione e così via. Il primo metodo trovato viene applicato. Dato che il programma compila, sappiamo che c'è almeno una versione del metodo m applicabile a obj.

Vediamo l'esempio di una variabile di tipo Bottiglia che, grazie a un Downcasting, a seconda delle circostanze dell'esecuzione del

programma può avere tipo esatto la classe Bottiglia oppure tipo esatto la classe BottigliaConTappo, più piccola.

```
//Bottiglia.java. Riutilizziamo il programma della Lezione 05
//BottigliaConTappo.java. Riutilizziamo il programma della Lezione 11

//TestCast.java   ESPERIMENTI SUL TIPO ESATTO DI UN OGGETTO
// tipo esatto di un oggetto = tipo C con cui l'oggetto x nasce
// x ha anche tipo (non esatto) ogni classe D che contiene C

public class TestCast {public static void main(String[] args)
{ // ESEMPIO. A seconda se "oggi_piove" e' vero o falso,
  // il tipo esatto di b e' la sottoclasse oppure la classe
   boolean oggi_piove = false;

// UPCAST: il passaggio a una classe superiore. E' sempre corretto
// basta dimenticare gli attributi aggiunti dalla sottoclasse
 Bottiglia b;
 if (oggi_piove) b = new BottigliaConTappo(10);
// upcast: b proviene da BottigliaConTappo e' spostato in Bottiglia
 else b = new Bottiglia(10);
// Se oggi_piove=true allora b si trova in BottigliaConTappo
// Se oggi_piove=true allora b non si trova in BottigliaConTappo

// DOWNCAST:  passaggio a una classe inferiore. Funziona SOLO
// nel caso in cui l'oggetto apparteneva GIA' alla classe
// inferiore ed e' stato spostato nella superiore da un upcast.

 // ESEMPIO. Il prossimo downcast appare corretto al compilatore
 // Java, il quale non ha modo di sapere se il tipo esatto di b e`
 // Bottiglia o BottigliaConTappo. A tempo di esecuzione viene
 // fatto un controllo sul tipo esatto di b e il downcast
 // fallisce (causando la terminazione anticipata del programma) se b
 // risulta avere tipo esatto Bottiglia

 BottigliaConTappo t = (BottigliaConTappo) b;
// SE b si trovava gia' in BottigliaConTappo ed e' stato spostato in
// Bottiglia allora il donwcast ha successo e scrivo:
 System.out.println( "Downcast avvenuto con successo");
// ALTRIMENTI il programma termina con una ClassCastException
```

```
// Dopo il downcast possiamo applicare a t un metodo aperta()
// che esiste solo nella sottoclasse BottigliaConTappo
System.out.println( "t.aperta() = " + t.aperta());
// Non possiamo scrivere b.aperta(), anche se nell'esecuzione b=t:
//     System.out.println( "b.aperta() = " + b.aperta());
// Il controllo di tipo del programma usa il tipo apparente Bottiglia
// di b, e nel tipo Bottiglia il metodo aperta non c'e'.
}}
```

Se non siamo sicuri se obj ha tipo esatto C o più piccolo, per evitare di sollevare una eccezione prima di scrivere ***((C) obj)*** dobbiamo fare il test ***(obj instanceof C)***. Solo se il test dà come risultato true possiamo eseguire ***((C) obj)***. Il test ***(obj instanceof C)*** vale vero se e solo se obj è un oggetto "istanziato" (cioè diverso da ***null***) di C. Come esempio, prendiamo un oggetto ***Bottiglia s=new BottigliaConTappo(10)***. Facciamo vedere che s è una istanza della classe ***BottigliaConTappo***, mentre null non lo è (anche se null ha tipo BottigliaConTappo).

```
// t instanceof T = true se e solo se
// t = oggetto istanziato (non null) di tipo T

public class InstanceOfDemo {public static void main(String[] args)
 {Bottiglia s = new BottigliaConTappo(10), t = new Bottiglia(10);
//s,t definiti da costruttore quindi !=null
  BottigliaConTappo u = null;
  System.out.println( "s instanceof BottigliaConTappo = " +
                      (s instanceof BottigliaConTappo)); // true
  System.out.println( "t instanceof BottigliaConTappo = " +
                      (t instanceof BottigliaConTappo)); // false
  System.out.println( "u instanceof BottigliaConTappo = " +
                      (u instanceof BottigliaConTappo)); // false
}}
```

Lezione 13. Parte 2. Un esempio di ereditarietà e di binding dinamico. Definiamo una classe ***Figura*** di figure, ciascuna con un suo metodo ***draw(Graphics g)*** che disegna la figura nella parte grafica g di una finestra. ***Disegno*** è la classe delle finistre con incluso un vettore di Figure. Noi forniamo un metodo ***void paint(Graphics g)*** per disegnare tutte le figure del vettore nella parte grafica g della finestra, con sistema di riferimento nel centro della finestra. ***Non definiamo il metodo***

draw per una figura generica, ma definiamo draw ogni volta che definiamo una sottoclasse di Figure.

```
//Graphics = a class of awt, consisting of graphical objects
//JFrame = a class of swing, including windows with
//border + title + close-iconify button
// FIGURA = oggetti con un metodo "draw" per disegnare una figura
// in un oggetto grafico g. Uguale all'unione di tutte le classi
// di figure. Ci consente di definire vettori di figure prese da
// classi diverse e di disegnarle usando un unico comando.

// Figura.java
import java.awt.*;    //Abstract Window Toolkit (finestre grafiche)
import javax.swing.*; //estensione di awt per interfaccie grafiche

public class Figura
{ // Dichiariamo il metodo di disegno draw ma lo definiamo vuoto:
  // serve solo per ricordarci di definire un metodo draw in ogni
  // sotto-classe della classe Figura.
    public void draw(Graphics g){ }}
```

Per la sottoclasse ***Quadrato*** di Figura, il metodo draw disegna un quadrato di lato dato, orizzontale e centrato con gli assi.

```
// Quadrato.java  quadrato=una possibile forma di una Figura
// Definiamo Quadrato come una sotto-classe di Figura
import java.awt.*;    //Abstract Window Toolkit
import javax.swing.*; //estensione di awt per interfaccie grafiche

public class Quadrato extends Figura
{ //Un quadrato e' definito dal suo lato
    private int lato;
  // COSTRUTTORE di un quadrato
    public Quadrato(int lato){ this.lato = lato; }

// OVERRIDE: RI-definiamo il metodo draw (per ora vuoto)
// per disegnare una figura nel caso di un quadrato.
// Scegliamo il quadrato centrato nell'origine e orizzontale
// Scegliamo il colore arancio per le prossime linee in g
 public void draw(Graphics g) {g.setColor(Color.orange);
 int m = lato / 2;
 g.drawLine( m,  m, -m,  m);   //disegno primo   lato su g
```

```
 g.drawLine(-m,  m, -m, -m);   //disegno secondo lato su g
 g.drawLine(-m, -m,  m, -m);   //disegno terzo   lato su g
 g.drawLine( m, -m,  m,  m);   //disegno quarto  lato su g
}}
```

Per la sottoclasse ***Cerchio*** di Figura, il metodo draw disegna un cerchio di raggio dato e centrato con gli assi.

```
// Cerchio.java cerchio = una possibile forma di una Figura:
// definiamo Cerchio come una sotto-classe di Figura
import java.awt.*;    //Abstract Window Toolkit
import javax.swing.*; //estensione di awt per interfaccie grafiche

public class Cerchio extends Figura
{//Un cerchio e' definito dal suo raggio r
    private int raggio;
 // COSTRUTTORE di un quadrato
    public Cerchio(int raggio){ this.raggio = raggio; }

//OVERRIDE: RI-definiamo il metodo draw per disegnare una figura
//in una finestra grafica g nel caso la figura sia un cerchio.
//Disegnamo il cerchio nel rettangolo di angolo in basso a sinistra
//(-r, -r) e di dimensioni 2r x 2r.
//Scegliamo il colore rosso per le prossime linee in g
 public void draw(Graphics g)
 {g.setColor(Color.red);
  g.drawOval(-raggio,-raggio,  2*raggio,2*raggio);}}


//Disegno.java
import java.awt.*;    //Abstract Window Toolkit (finestre grafiche)
import javax.swing.*; //estensione di awt per interfaccie grafiche

public class Disegno extends JFrame /* Un "disegno" e' un JFrame con
parte grafica = tutte le figure di un vettore di figure */
{private Figura[] figure;
 //COSTRUTTORE basato sul costruttore della classe superiore JFrame
 public Disegno(Figura[] figure)
{super();               //Assegnamo tutti i parametri di un JFrame
 this.figure = figure;  //Aggiungiamo un vettore di figure
}
```

```
// OVERRIDE: ridefiniamo il metodo "paint" di JFrame
// chiedendo di inizializzare una finestra grafica, poi
// di disegnare tutte le figure del vettore "figure" in g

public void paint(Graphics g) {
int w = getSize().width;  // base    frame g
int h = getSize().height; // altezza frame g
g.clearRect(0, 0, w, h);  // azzero  contenuto del frame g
g.translate(w/2,h/2); //translo sistema di riferimento a centro frame

//DISEGNO tutte le figure del vettore "figure"
for(int i=0;i<figure.length;++i) figure[i].draw(g);
//BINDING per il metodo draw. Quale metodo draw viene scelto?
//Dipende dal tipo esatto di figura[i]. Se figura[i] ha tipo esatto
//Quadrato, allora viene scelto il metodo draw per i quadrati e non
//il metodo draw per le figure (che sarebbe un metodo vuoto)
}

public static void main(String[] args)
{int m=70,n=90; Figura[] figure = new Figura[n];
//Vettore di n figure: non scegliamo ancora quali

//Assegnamo le n figure: scegliamo m quadrati e (n-m) cerchi
//Possiamo farlo perche' quadrati e cerchi sono particolari figure
 for(int i = 0; i<m; i++) figure[i] = new Quadrato(i*7);
 for(int i = m; i<n; i++) figure[i] = new Cerchio(i*3);

//Definiamo un disegno con vettore di figure proprio "figure"
 Disegno frame = new Disegno(figure);//Jframe con vettore di figure

//ESEMPI DI EREDITARIETA' (SENZA OVERRIDE)
//Scegliamo di terminare la figura quando ne chiudiamo la finestra
//(il metodo setDefaultCloseOperation viene ereditato da JFrame)
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//Scegliamo la dimensione della finestra grafica:
//(il metodo setSize viene ereditato da JFrame)
 frame.setSize(600,600);
//Rendo il disegno visibile, con n figure disegnate da "paint":
//(il metodo setVisible viene ereditato da JFrame)
 frame.setVisible(true);}}
```

Il risultato: una finestra contenente un disegno fatto di: 70 quadrati arancio e 20 cerchi rossi, concentrici

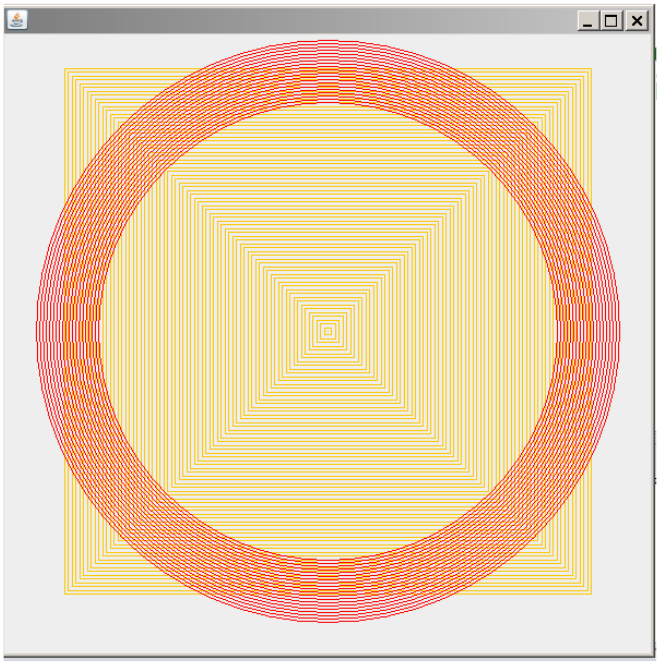
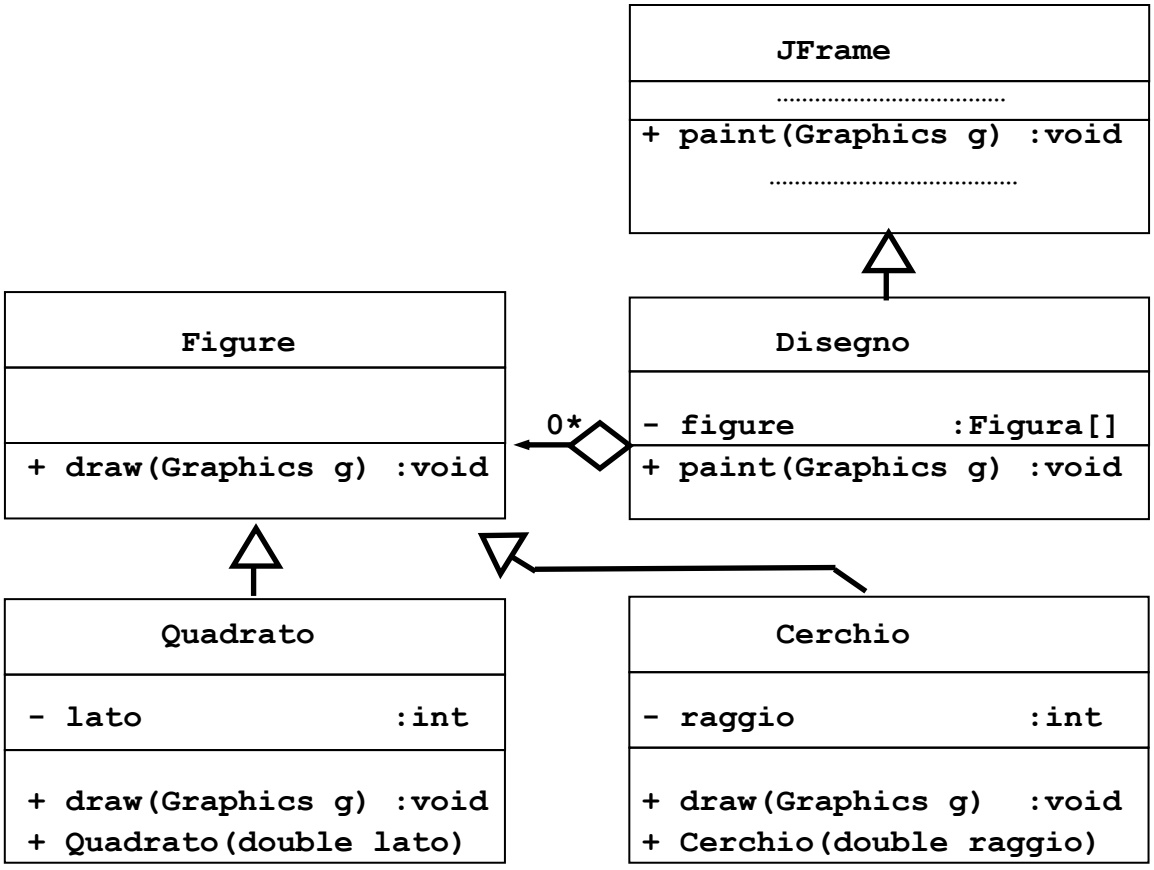


Diagramma UML della classe Disegno

(omettiamo di indicare il main di Disegno)

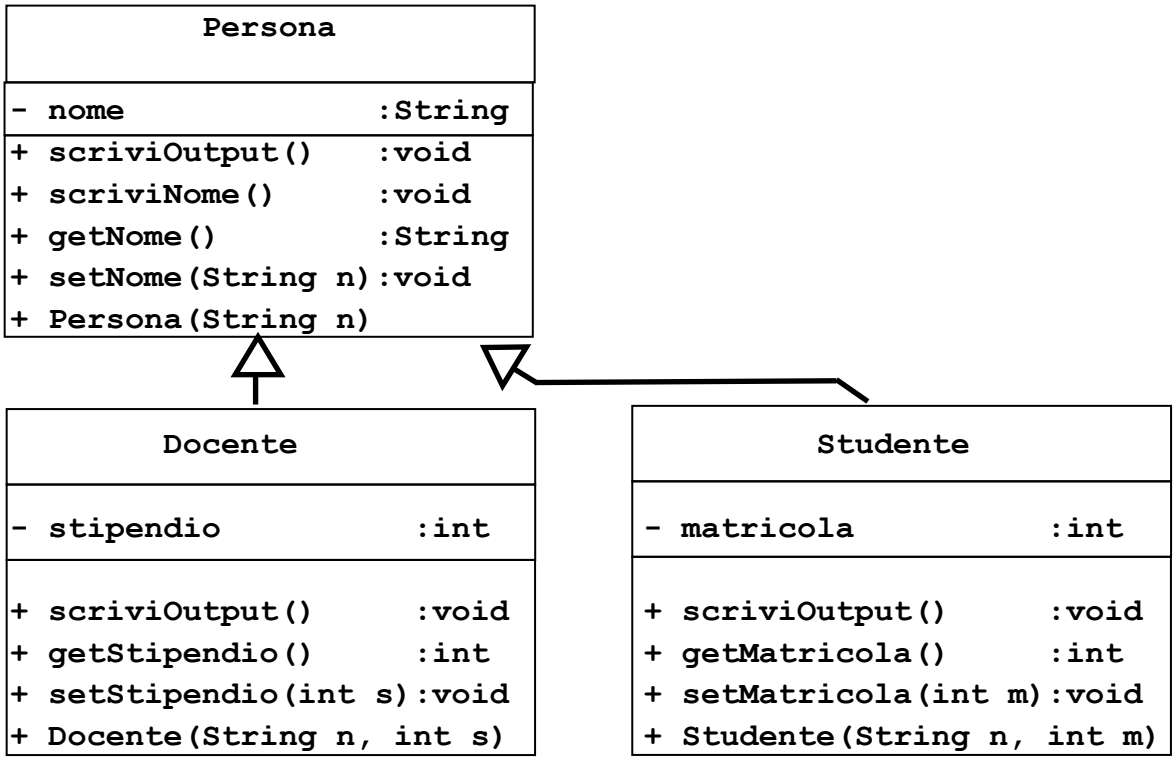


Lezione 14 Esempi di ereditarietà e vettori come liste

Lezione 14. Parte 1. Esempio di uso del tipo esatto per l'ereditarietà e l'override.

Nella classe ***Persona*** inseriamo due metodi uguali di scrittura, ***void scriviOutput()*** e void ***scriviNome()***. Il primo viene riscritto in ogni sottoclasse, il secondo no. Quando applichiamo i due metodi a un vettore di persone nel primo caso il metodo usato è quello della sottoclasse a cui appartiene un dato oggetto, nel secondo caso è quello della classe Persona.

Diagramma UML della classe Persona




```
//Persona.java
public class Persona
{ private String nome;
  public  Persona(String nome)        {this.nome=nome;}
  public  String getNome()            {return nome;}
  public  void   setNome(String nome){this.nome=nome;}

//Metodo di scrittura di cui faccio OVERRIDE in ogni sottoclasse:
//si arricchisce man mano che ci sono piu' informazioni su un oggetto
  public void scriviOutput()
   {System.out.println( " nome = " + nome);}
```

```
//Metodo di scrittura di cui NON intendo fare OVERRIDE
//resta uguale anche se quando sono informazioni in piu'
  public void scriviNome()
   {System.out.println( " nome = " + nome);}
//Ma per ora i due metodi di scrittura sono uguali!
}

//Docente.java
public class Docente extends Persona
{
//Su un docente abbiamo delle informazioni in piu' che su una persona
  private int stipendio;
  public Docente(String nome, int stipendio)
    {super(nome); this.stipendio = stipendio;}
  public int getStipendio(){return stipendio;}
  public void setStipendio(int stipendio)
    {this.stipendio = stipendio;}

//OVERRIDE del metodo scriviOutput(): stampo le informazioni in piu'
  public void scriviOutput()
   {super.scriviOutput();
     System.out.println( " stipendio = " + stipendio);}

//NON faccio override di scriviNome():
//questo metodo viene semplicemente ereditato
}

//Studente.java
public class Studente extends Persona
{ //Su uno studente abbiamo informazioni in piu' che su una persona
 private int matricola;
  public Studente(String nome, int matricola)
    {super(nome);this.matricola = matricola;}
  public int getMatricola(){return matricola;}
  public void setMatricola(int matricola){this.matricola =
matricola;}

//OVERRIDE del metodo scriviOutput(): stampo le informazioni in piu'
  public void scriviOutput()
   {super.scriviOutput();
     System.out.println( " matricola = " + matricola);}
```

```
//NON faccio override di scriviNome(): questo metodo
//viene semplicemente ereditato
}
```

Quando applichiamo ***void scriviOutput()*** e ***void scriviNome()*** a docenti e studenti nella classe Persona, nel primo caso il metodo usato è quello della sottoclasse Docente o Studente, nel secondo caso è quello della classe Persona.

```
//PersonaDemo.java
public class PersonaDemo{public static void main(String[] args){
//Definisco delle persone appartenenti a sottoclassi
 Studente a = new Studente( "Rossi",111);   //111=matricola
 Docente  b = new Docente( "Ferrero",1000); //1000= stipendio
//Definisco un vettore con le persone appena introdotte
 int n=2; Persona[] c = new Persona[n]; c[0]=a;c[1]=b;
//tipo apparente c[0],c[1]: Persona, tipo esatto: Studente, Docente

//Stampo c usando il metodo scriviOutput() (CON override): Java
//utilizza il tipo "esatto" (il tipo originario) di ogni oggetto
//il metodo scriviOutput() per il tipo esatto
 System.out.println( "\nEsempio di scriviOutput()");
 for(int i=0;i<n;i++)
 {System.out.println(i + " ");c[i].scriviOutput();}

//Stampo c usando il metodo scriviNome() (SENZA override):
//Java utilizza il tipo esatto di ogni oggetto e il metodo
//scriviNome() per il tipo esatto. IN QUESTO CASO IL METODO E'
//EREDITATO E RESTA SEMPRE LO STESSO (no override)

  System.out.println( "\nEsempio di scriviNome()");
  for(int i=0;i<n;i++)
  {System.out.println(i + " ");c[i].scriviNome();}  }}
```

Lezione 14. Parte 2. Le classi MiniLinkedList e Iterator. Definiamo la classe ***MiniLinkedList*** delle liste concatenate V, attraverso una lista di nodi V_0, ..., V_(size-1), ognuno dei quali punta al successivo, e con un attributo ***size*** che indica il numero dei nodi. La lista viene identificata con l'indirizzo ***first*** di V_0. I metodi pubblici di MiniLinkedList sono

1. ***int get(int i)*** Restituisce il contenuto del nodo numero i se 0≤i<size. *Con accesso lento, numero di getNext() richiesti = i.*
2. ***void set(int i, int x)*** Assegna il valore x al nodo numero i, se 0≤i<size.
3. ***void add(int i, int x)*** Aggiunge un nodo di contenuto x in posizione i, se 0≤i≤size, e incrementa size.
4. ***int remove(int i)*** che cancella il nodo di posto i dalla lista, se 0≤i<size, e decrementa size.

MiniLinkedList assomiglia alla classe dei vettori di dimensioni statiche, con il vantaggio che la dimensione size di V non è fissata a priori, e che V occupa esattamente lo spazio di memoria di cui ha bisogno. C'è però uno svantaggio: l'operazione V.get(i) raggiunge il nodo i passando attraverso i nodi 0, …, i-1, quindi usa un numero di getNext() uguale ad i. L'accesso a un vettore statico, invece, avviene con un solo accesso a un indirizzo. Quindi passare una volta attraverso ciascun nodo della lista richiede **0+1+2+ … + (size-1)** volte getNext(). Dunque in totale richiede **size(size-1)/2** volte getNext(), un costo *inaccettabile* per grandi valori di size.

Vogliamo consentire un passaggio *veloce* in sola lettura a tutti i nodi della lista, con un numero totale di getNext() pari a size.

Un modo semplice è rendere ***pubblico*** l'attributo first della lista, e usare un ciclo che rimpiazzi ogni nodo con il successivo fino a esaurimento nodi. Questo però comporta il ***rischio*** che un programmatore possa modificare la lista dall'esterno, e che per esempio si dimentichi di modificare l'attributo size della lista quando aggiunge/toglie nodi.

Per consentire un passaggio veloce ma senza consentire di modificare i nodi (tranne che con i metodi di MiniLinkedList) introduciamo la classe ***Iterator*** dei puntatori ai nodi all'interno di una lista. Iterator ha un metodo ***int next()*** che legge il contenuto di un puntatore e sposta il puntatore al nodo dopo, ma ***non consente di modificare i nodi***. Iterator viene usata per iterare la stessa operazione (per esempio: *la stampa*) su tutti gli elementi di una lista, senza consentire di modificare i nodi della lista. Per farlo occorre usare i metodi di ***MiniLinkedList***, che modificano l'attributo size se la dimensione della lista cambia.

//Node.java Riutilizziamo la classe Node della Lezione 08

```
//MiniLinkedList.java
public class MiniLinkedList
{private Node first; private int size;
// INVARIANTE DI CLASSE:(first=elemento n.0 della lista concatenata)
// e (size = numero nodi accessibili da first)

//Costruttore della lista vuota con 0 elementi
 public MiniLinkedList(){first = null; size = 0;}
 public int size() {return this.size;}

//Metodo privato node(i) = indirizzo del nodo V_i della lista V.
//Viene usato dalla classe per definire gli altri metodi
//Non viene reso pubblico per evitare che dall'esterno sia
//possibile modificare i nodi della classe senza aggiornare size

 private Node node(int i){//controllo che V_i sia un nodo della lista
  assert 0 <= i && i < size;
//creo una copia di first per non modificare l'originale
  Node p = this.first;
  while (i > 0) //rimpiazzo per i volte: p con il nodo dopo
  {assert p != null; //se vale l'invariante questo assert e' vero
   p = p.getNext(); i--;}
//Dopo aver applicato p = p.getNext() per i volte abbiamo p=node(i)
  assert p != null; //se vale l'invariante questo assert e' vero
  return p;}

//DEFINIZIONE get(i), set(i,x), add(i,x), remove(i) usando node(i)
//get(i) = contenuto node(i)
  public int get(int i){return node(i).getElem();}

//set(i,x) assegna node(i) ad x
  public void set(int i, int x){node(i).setElem(x);}

//add(i,x) aggiunge un nodo che contiene x in posizione i
 public void add(int i, int x) {
 assert 0 <= i && i <= size;
 if (i == 0) {first = new Node(x, first);} //aggiungo un nodo
all'inizio
 else { //calcolo il nodo precedente al nodo da aggiungere
  Node prev = node(i - 1);
//aggiungo un nodo tra prev e prev.getNext()
```

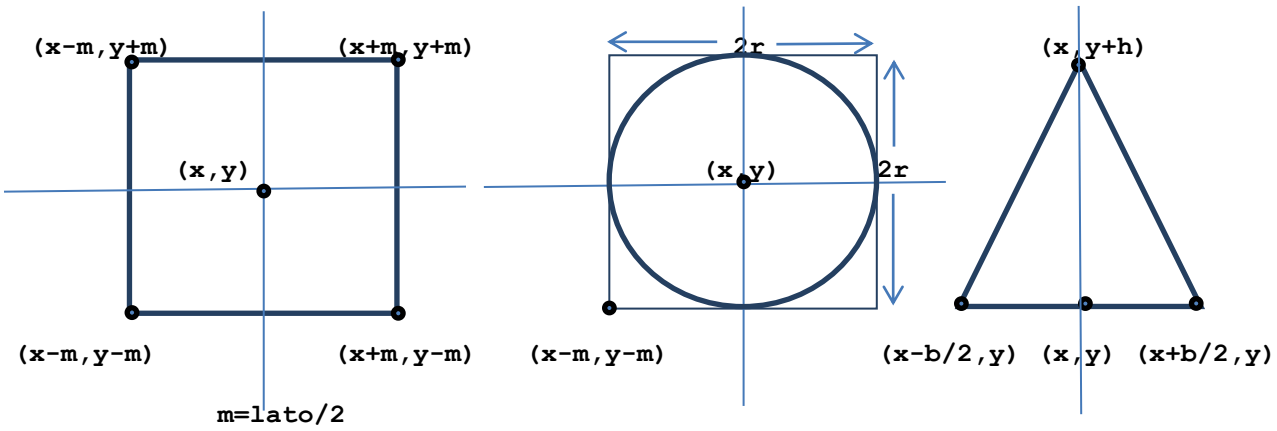
```
  prev.setNext(new Node(x, prev.getNext()));}
// l'invariante di classe e` temporaneamente non valido: size vale
// uno meno il numero di elementi della lista. Quindi aggiungiamo 1
 size++;}

//remove(i) elimina il nodo n. i e ne restituisce il contenuto x
 public int remove(int i) {assert 0 <= i && i < size; int x;
 if (i == 0) { //Elimino first. Nuovo first = vecchio first.getNext()
  x = first.getElem();
  first = first.getNext();}
  else //i>0
  {//Nodo prev precedente il nodo da eliminare = node(i-1)
   Node prev = node(i - 1);
//Nodo el da eliminare = nodo che viene dopo prev
    Node el = prev.getNext(); x = el.getElem();
//Per eliminare el, collego prev al nodo che viene dopo el
    prev.setNext(el.getNext());}
// L'invariante di classe e` temporaneamente non valido: size vale
// uno piu' il numero di elementi della lista. Dobbiamo sottrarre 1
  size--; return x;}

//Definiamo un metodo che ridenomina una MiniLinkedList in un
//elemento della classe MiniIterator. MiniIterator consente di
//eseguire un ciclo for senza rendere pubblici gli indirizzi dei nodi
  public MiniIterator iterator()
  {return new MiniIterator(first);}}

//MiniIterator.java. Classe che consente di traversare una lista
//con un numero "size" di applicazioni di getNext()
//senza rendere pubblici gli indirizzi dei nodi
public class MiniIterator {
private Node next; // next = prossimo nodo da "visitare"
public MiniIterator(Node first) {next = first;}
public boolean hasNext()
{return next != null;}
public int next()
{assert hasNext();
int x = next.getElem();
next = next.getNext();
return x;}}

//TestMiniIterator.java (controlliamo MiniLinkedList e MiniIterator)
```

```
public class TestMiniIterator
{ public static void main(String[] args) {
//Definisco una lista l = {9,8,7,6,5,4,3,2,1,0} aggiungendo
//              0,1,2,3,4,5,6,7,8,9
//sempre in posizione 0, dunque ogni elemento davanti ai precedenti
 MiniLinkedList l = new MiniLinkedList();
 for (int i = 0; i < 10; i++) l.add(0, i);

//Cancello l_7, cioe' il terzo elemento di l dal fondo: il 2
 System.out.println( "l.getSize()  = " + l.getSize());
 System.out.println( "Cancello l_7 = 3");
 System.out.println( "l.remove(7)  = " + l.remove(7));
 System.out.println( "l.getSize()  = " + l.getSize());
//Stampo usando la classe MiniIterator per il puntatore della stampa
 MiniIterator i = l.iterator();
 while (i.hasNext())
  System.out.println(i.next());}}
```

Esercitazione 03 Vettori di figure

Riprendete il disegno di quadrati e cerchio in un ***Jframe*** (in una finestra Java) visto nella Lezione 13. Ricopiate tutto il codice, quindi modificate le classi Quadrato e Cerchio aggiungendo attributi privati per le coordinate (intere) x, y del ***centro*** del quadrato e del centro del cerchio, e per un ***colore*** c (cercate informazioni sulla classe ***Color*** di Java). Aggiungete una classe Triangolo dei triangoli isosceli, descritti da base, altezza, coordinate x,y del ***punto medio*** della base (tutti interi, col segno), e colore. Usate queste classi per costruire una finestra Java con disegnato un vettore contenente quadrati, cerchi, triangoli, di differente centro, dimensione e colore. I costruttori delle classi avranno quindi argomenti:

Quadrato(x,y,l,c), Cerchio(x,y,r,c), Triangolo(x,y,b,h,c)



Nota. La classe Color si trova nella libreria ***java.awt***, e viene caricata dagli stessi comandi (visti nella Lezione 13) che caricano la classe Jframe delle finestre grafiche:

```
import java.awt.*;    //Abstract Window Toolkit (finestre grafiche)
import javax.swing.*; //estensione di awt per interfaccie grafiche
```

Un colore si definisce con ***Color.nome*** (avete a disposizione i nomi: black, red, green, yellow, blue …) oppure con ***new Color(r,g,b)***, dove r,b,g sono interi da 0 a 255 che esprimono le proporzioni di rosso, verde e blu nel colore. Su Wikipedia potete trovate per ogni colore i valori di r,g,b (da 0 a 255) necessari per definirlo.

Soluzione Esercitazione 03

```
// Figura.java
import java.awt.*;    //Abstract Window Toolkit
import javax.swing.*; //estensione awt per interfaccie grafiche
public class Figura{public void draw(Graphics g){ }}

//Quadrato.java
import java.awt.*;    //Abstract Window Toolkit
import javax.swing.*; //estensione di awt per interfaccie grafiche

public class Quadrato extends Figura
{private int x;private int y; private int lato;private Color c;

//Costruttore
public Quadrato(int x, int y, int lato, Color c)
{this.x=x;this.y=y;this.lato = lato;this.c=c;}

public void draw(Graphics g){g.setColor(c);
   int m = lato / 2;
   g.drawLine( x+m, y+m, x-m, y+m);   //disegno primo   lato su g
   g.drawLine( x-m, y+m, x-m, y-m);   //disegno secondo lato su g
   g.drawLine( x-m, y-m, x+m, y-m);   //disegno terzo   lato su g
   g.drawLine( x+m, y-m, x+m, y+m);   //disegno quarto  lato su g
}}

//Cerchio,java
import java.awt.*;    //Abstract Window Toolkit
import javax.swing.*; //estensione di awt per interfaccie grafiche

public class Cerchio extends Figura
{private int x;private int y;private int raggio;private Color c;

//COSTRUTTORE
public Cerchio(int x, int y, int raggio, Color c)
{this.x = x;this.y = y;this.raggio = raggio;this.c = c;}

public void draw(Graphics g){g.setColor(c);
 g.drawOval(x-raggio,y-raggio,  2*raggio,2*raggio);}}

//Triangolo.java
import java.awt.*;    //Abstract Window Toolkit
import javax.swing.*; //estensione di awt per interfaccie grafiche
```

```
public class Triangolo extends Figura
{private int x;private int y;private int b;private int h;
 private Color c;

//COSTRUTTORE
public Triangolo(int x, int y, int b, int h, Color c)
{this.x = x;this.y = y;this.b = b; /*base*/ this.h = h; /*altezza*/
 this.c = c;}

public void draw(Graphics g){g.setColor(c);
int m=b/2;
g.drawLine(x-m, y, x+m, y);   //base triangolo
g.drawLine(x-m, y, x, y+h);
g.drawLine(x+m, y, x, y+h);}}

//Disegno.java
import java.awt.*;    //Abstract Window Toolkit (finestre grafiche)
import javax.swing.*; //estensione di awt per interfaccie grafiche
import java.util.Random; //per i numeri casuali

public class Disegno extends JFrame
{private Figura[] figure;

//COSTRUTTORE
public Disegno(Figura[] figure){
 super();               //Assegnamo tutti i parametri di un JFrame
 this.figure = figure; //Aggiungiamo un vettore di figure
    }

public void paint(Graphics g){
 int w = getSize().width;  // base    frame g
 int h = getSize().height; // altezza frame g
 g.clearRect(0, 0, w, h);  // azzero  contenuto del frame g
 g.translate(w/2,h/2);//translo sistema di riferimento a centro frame

 for(int i=0;i<figure.length;++i) figure[i].draw(g);}

 public static Random random = new Random();
 public static int v()     {return random.nextInt(255);}
 public static Color c()   {return new Color(v(),v(),v());}
 public static int p()     {return random.nextInt(400)-200;}
```

```
public static void main(String[] args){
 int n=20;int i;
 Figura[] figure = new Figura[3*n];
 for(i=0;i<n;++i)figure[i]=new Quadrato(p(),p(),p(),c());
 for(i=n;i<2*n;++i)figure[i]=new Cerchio(p(),p(),p(),c());
 for(i=2*n;i<3*n;++i) figure[i]=new Triangolo(p(),p(),p(),p(),c());

 Disegno frame = new Disegno(figure);
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frame.setSize(600,600);
 frame.setVisible(true);}}
```

Lezione 15 Classi astratte di figure

Lezione 15. Una classe Figure per il calcolo di area e perimetro. Nella Lezione 13 visto come risolvere il problema di ***disegnare*** figure diverse: quadrati, cerchi, triangoli, usando la nozione di ***sottoclasse***. In questa lezione consideriamo un problema simile: supponiamo di voler definire delle classi per ***calcolare area e perimetro*** di diverse figure geometriche: cerchi, poligoni regolari, trapezi, rettangoli eccetera. Vediamo una versione migliorata della stessa soluzione, ***dichiarando*** una classe Java come ***classe astratta.***

Una prima soluzione usando la nozione di sottoclasse. Dobbiamo scrivere una copia di ogni metodo statico su figure per i cerchi, i poligoni regolari eccetera. Non possiamo fare un vettore di figure perché le figure non hanno un tipo comune. Avremmo quindi bisogno di una sola classe Figura che le contenga tutte le classi citate, e che non contenga oggetti oltre agli oggetti delle sottoclassi e a ***null***. La difficoltà è che non esiste un metodo generale per calcolare area e perimetro di una figura: dobbiamo quindi dichiarare dei metodi vuoti per le figure e sovrascriverli per ogni sottoclasse. Questa soluzione adatta la soluzione vista nella Lezione 13.

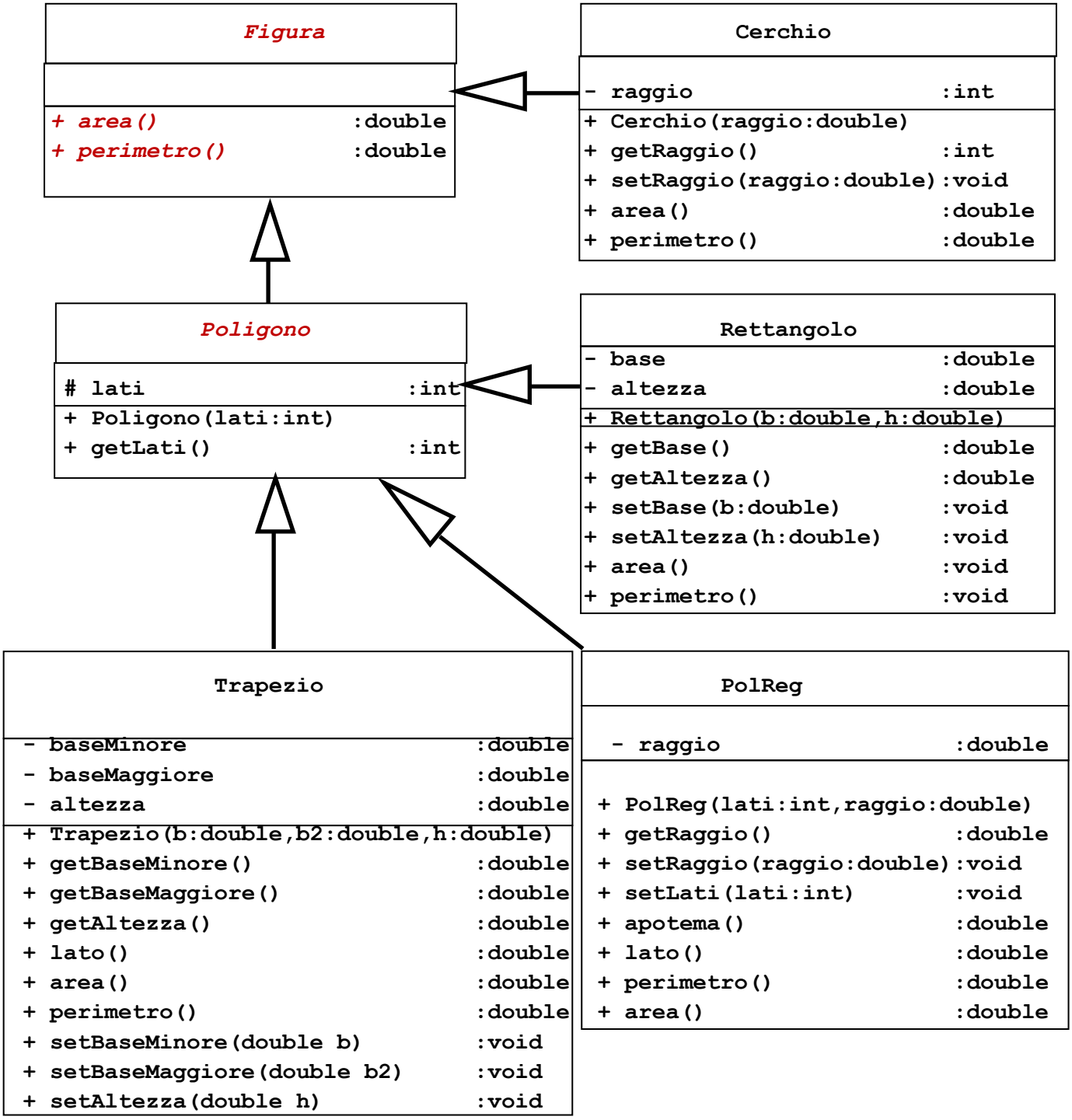
Una soluzione migliorata: dichiariamo in Java una classe come astratta. Possiamo migliorare la soluzione precedente dichiarando una classe e alcuni dei suoi metodi **astratti**. I metodi astratti fanno solo da segnaposto per i metodi che si trovano in classi più piccole, non si possono usare. E non possiamo costruire oggetti in una classe astratta, a parte il ***null***. Il **vantaggio** di dichiarare una classe astratta è che Java controlla per me che io non usi un metodo astratto prima di averlo sovrascritto, e che io non costruisca oggetti in una classe astratta.

Regole per le classi astratte. Gli attributi non sono astratti. Se un metodo e' astratto la sua classe e' astratta. Una classe astratta non ha costruttori. Una classe astratta contiene null e gli oggetti definiti nelle sottoclassi non astratte. Perché una sottoclasse di una classe astratta non sia astratta deve sovrascrivere tutti i metodi astratti che eredita.

Nei diagrammi UML, operazioni e classi astratte sono indicate scrivendone il nome in ***corsivo.***

Diagramma UML della classe astratta Figura e sottoclassi

(omettiamo di indicare i main di prova)



```
//Figura.java

//LA CLASSE ASTRATTA FIGURA
public abstract class Figura
//Cerchi, Poligoni regolari, Trapezi, Rettangoli, ...
{ //Non siamo obbligati a fornire attributi e costruttori per Figura

//METODI ASTRATTI per area e perimetro: si possono usare solo
//quando vengono sovrascritti in una sottoclasse
    public abstract double area();
    public abstract double perimetro();

//Se f e' un cerchio, rettangolo eccetera, assegno a f tipo Figura e
//ne calcolo l'area scrivendo f.area(), anziche' dover considerare
//un caso diverso per ogni tipo di figura.

// ESEMPIO di cosa viene accettato da Java.
// posso definire l'oggetto null

  public static void main(String[] args)
  {Figura P; //OK: definisce null
   }}

// Cerchio.java Sottoclasse non astratta di Figura:
// sovrascrive area() e perimetro(),
// ha un attributo e metodi specifici per il raggio del cerchio

public class Cerchio extends Figura
{private double raggio;

 public Cerchio(double raggio)
 {assert raggio >= 0;this.raggio = raggio;}

 public double getRaggio(){return raggio;}
 public void setRaggio(double raggio)
 {assert raggio >= 0;this.raggio = raggio;}

    public double area()
    { return Math.PI * raggio * raggio; }

    public double perimetro(){return 2 * Math.PI * raggio; }}
```

```
// Poligono.java. Sottoclasse astratta di Figura:
// anche se piu' piccolo di Figura, non ha
// dei metodi veri per calcolare area e perimetro,
// ha degli attributi e un metodo di lettura per il numero dei lati.
// Possiede sottoclassi non astratte: PolReg, Triangolo,...

public abstract class Poligono extends Figura
{protected int lati; /* "protected" consente di modificare "lati" in
una sottoclasse di Poligono (se cosi' vogliamo) */
 public Poligono(int lati)
 { //Non e' necessario invocare il costruttore della classe superiore
   //quando e' il costruttore di default, quindi omettiamo:
   //super();
 assert lati >= 3;this.lati = lati;}

 public int getLati(){ return lati; }
 // In alcune sottoclassi il numero dei lati puo' cambiare
 // ma in altre no, quindi per ora niente metodo set

// ESEMPIO di cosa viene accettato o no da Java.
// posso definire l'oggetto null in Poligoni
// ma non ci sono oggetti di tipo esatto la classe astratta

  public static void main(String[] args)
  {Poligono P; //OK: definisce null

//Poligoni triangolo = new Poligoni(3); //NON corretto:
//"triangolo" e' costruito nella classe astratta Poligono
  }
}

/* PolReg.java
Sottoclasse non astratta di Poligono e quindi di Figura: ha dei
metodi veri per area e perimetro, piu' attributi e metodi specifici
per raggio, lato e apotema. NON e' possibile compilare la classe
PolReg finche' non si sovrascrivono i metodi astratti
area() e perimetro() */

public class PolReg extends Poligono
{private double raggio;
 public PolReg(int lati, double raggio)
  {super(lati); assert raggio >= 0; this.raggio = raggio;}
```

```
//Il numero dei lati di un poligono regolare puo' cambiare.
public double getRaggio(){ return this.raggio; }
public void setRaggio(double raggio){ this.raggio=raggio; }
public void setLati(int lati){ this.lati=lati; }

//Formula per apotema
public double apotema()
{ return raggio * Math.cos(Math.PI / getLati()); }

//Formula per lato
public double lato()
{ return 2 * raggio * Math.sin(Math.PI / getLati()); }

//Formula per perimetro
public double perimetro()
{ return lato() * getLati(); }

//Formula per area
public double area()
{ return getLati() * (lato() * apotema() / 2); }

/* ESEMPIO di cosa viene accettato da Java. Questo main serve a far
vedere che nella classe astratta Figura posso definire un oggetto E
proveniente dalla sottoclasse PolReg, perche' questa sottoclasse non
e' astratta. Il tipo esatto di E determina il metodo usato per
calcolare l'area di E. */

 public static void main(String[] args)
  {Figura E = new PolReg(6,1.0);
   System.out.println(E.area()); //Area calcolata nella classe PolReg
  }
}

// Trapezio.java   (Trapezio Isoscele)

//Sottoclasse non astratta di Poligono: sovrascrive i metodi per
//calcolare area e perimetro, in piu' ha attributi e metodi specifici
//per i trapezi isosceli: base maggiore, minore, altezza
//Il numero dei lati e' fisso a 4.
public class Trapezio extends Poligono
{private double baseMinore;
 private double baseMaggiore;
```

```
 private double altezza;

    public Trapezio
     (double baseMinore, double baseMaggiore, double altezza)
    { // l'invocazione a super deve
      // essere la prima istruzione del
      // costruttore
 super(4); //Il trapezio e' un poligono di 4 lati
 assert baseMinore > 0 && baseMinore <= baseMaggiore && altezza > 0;
 this.baseMinore = baseMinore;
 this.baseMaggiore = baseMaggiore;
 this.altezza = altezza;}

 public double getBaseMinore()  { return baseMinore; }
 public double getBaseMaggiore(){ return baseMaggiore; }
 public double getAltezza()     { return altezza; }

 public void setBaseMinore(double baseMinore)
 { this.baseMinore = baseMinore; }
 public void setBaseMaggiore()
 { this.baseMaggiore = baseMaggiore;  }
 public void setAltezza(double altezza)
 { this.altezza=altezza; }

//Formula per l'area del trapezio (anche non isoscele)
public double area()
{return (baseMinore + baseMaggiore) * altezza / 2;}

//Formula per il perimetro del trapezio isoscele
public double perimetro()
{return 2 * lato() + baseMinore + baseMaggiore;}

//Formula per il lato del trapezio isoscele
 public double lato()
 {return Math.sqrt(Math.pow(altezza, 2)
       + Math.pow((baseMaggiore - baseMinore) / 2, 2));}}

//Rettangolo.java
// Sottoclasse non astratta di Poligono e quindi di Figura.
// Rettangolo ha dei metodi "veri", non astratti, uno ereditato dalla
// classe Poligono: getLati(). Il numero dei lati e' fisso a 4.
```

```
public class Rettangolo extends Poligono
{private double base; private double  altezza;
 public Rettangolo(double base, double altezza)
 {super(4);this.base=base;this.altezza=altezza;}
 public double getBase()                {return base;}
 public double getAltezza()             {return altezza;}
 public void setBase(double base)       {this.base=base;}
 public void setAltezza(double altezza){this.altezza=altezza;}
 public double area()                   {return base*altezza;}
 public double perimetro()              {return 2*(base+altezza);}
}

//TestFigura.java
/* Classe introdotta per provare a usare la classe Figura e il "late
binding". La classe Figura ha solo metodi astratto per area e
perimetro, non utilizzabili. Gli oggetti di Figura appartengono tutti
a sottoclassi non astratte, dove esistono metodi che sovrascrivono
area() e perimetro(): sono questi ultimi ad essere usati. Vediamo un
esempio. */

public class TestFigura
{ /* Metodo per trovare la figura di massima area in un vettore di
figure non vuoto */
 public static int maxArea(Figura[] V)
 {
 // esempio di LATE BINDING (detto anche "binding dinamico")
 // metodo da eseguire: determinato dal tipo esatto di un oggetto
  int n = V.length;
  assert n>0; //controllo che a non sia vuoto
  int m=0; //m=indice massima area trovata, all'inizio indice di V[0]
  for(int i=1;i<n;i++) {if (V[i].area()>V[m].area()) m=i;}
 //ogni volta che trovo un'area V[i] piu' grande di V[m] aggiorno m
  return m;
 }

  public static void main(String[] args)
 {Figura f = new Cerchio(1.0);
/* Il tipo esatto di f e' il cerchio, quindi l'area di f si calcola
con il metodo area() definito per i cerchi */
 System.out.println( "\nArea cerchio f di raggio 1 = " + f.area());

//Finalmente incassiamo un vantaggio dalla classe astratta Figura:
```

```
//ora posso definire un vettore con figure di OGNI TIPO e gestirlo
//con un solo metodo statico maxArea
 Figura[] a = {new Cerchio(1.0),     // Cerchio di raggio 1
  new Rettangolo(1,2),  // Rettagolo di base 1 e altezza 2
  new PolReg(6, 2),     // Esagono regolare di raggio 2
  new Trapezio(1, 2, 3) // Base minore 1, base maggiore 2, altezza 3
 };

int n = a.length;
System.out.println( "\nStampo area e perimetro delle figure in a.");
System.out.println( "Non ottengo il valore esatto 12 del perimetro
dell'esagono.");
 for(int i=0;i<n;i++)
 {System.out.println(" Area a[" + i + "] = " + a[i].area());
  System.out.println(" Perimetro a[" + i + "] = " +
                      a[i].perimetro());}

System.out.println("\nIndice figura massima area in a="+ maxArea(a));
}}
```

Lezione 16 La classe astratta Lista

Lezione 16. Le liste come una classe astratta ricorsiva. Definiamo una classe astratta ***List*** delle ***liste ordinate senza ripetizioni di interi***. List ha due sottoclassi concrete, la classe ***Nil*** che contiene (oltre a null) la sola lista vuota, e la classe ***Cons*** delle liste contenenti un elemento "elem" e una lista "next" di elementi tutti più grandi di elem. Usiamo List per rappresentare insiemi finiti di interi. Usare liste ordinate senza ripetizioni ha il vantaggio di avere una sola rappresentazione per insieme.

Il primo passo è scegliere quali metodi implementare (tutti dinamici e pubblici). Chiediamo di avere un metodo ***boolean empty()*** per decidere se una lista è vuota, un metodo ***int size()*** per contare gli elementi di una lista, un metodo ***boolean contains(int x)*** per decidere se una lista contiene un elemento x, e due metodi per costruire nuove liste ordinate senza ripetizioni. Il primo è ***List insert(int x)*** che crea una nuova lista aggiungendo un elemento x a una lista data, dopo gli elementi più piccoli e prima di quelli più grandi, e non aggiunge x se x c'è già. Il secondo è ***List append(List l)*** che costruisce una nuova lista unione della lista data e della lista l. In questo esercizio non definiamo metodi che modificano liste già esistenti.

Il costruttore ***Cons(int elem, List next)***, della classe Cons di liste non vuote, consente di definire liste non ordinate: per questo motivo non ne consentiamo l'uso fuori di List e lo dichiariamo ***protected***. Non possiamo definire Cons come ***private***, perché Cons viene usato nella classe Nil e dunque fuori di Cons.

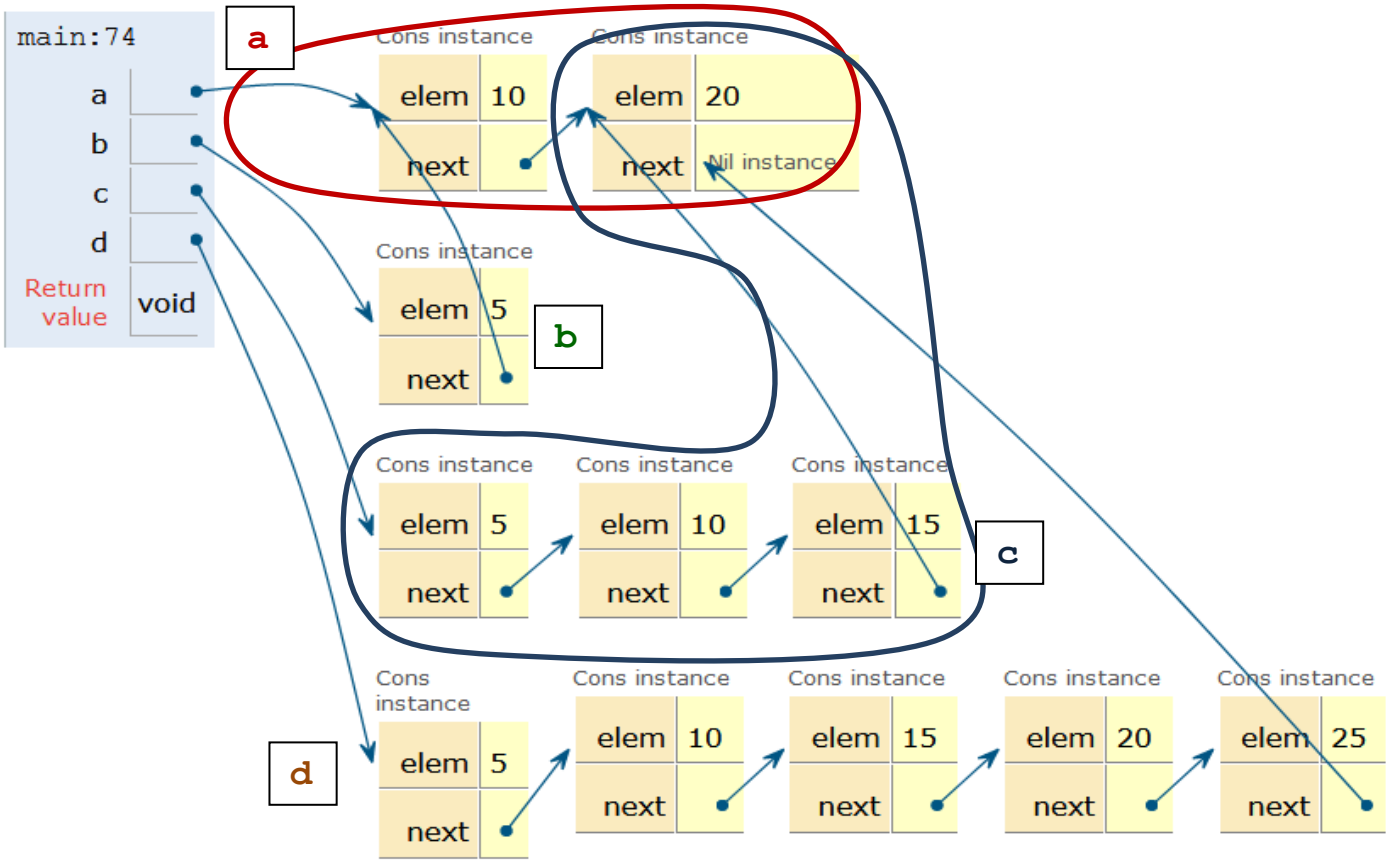
Un'ultima scelta di implementazione è la seguente. Quando definiamo una nuova lista (ordinata e senza ripetizioni), vogliamo riutilizzare per quanto possibile la lista da cui partiamo. Siano

```
a    = {10,20}
b    = a.insert(5)  = {5,10,20}
c    = b.insert(15) = {5,10,15,20}
d    = c.insert(25) = {5,10,15,20,25}
```

Vogliamo che insert costruisca le liste b,c,d riutilizzando per quando possibile le liste precedenti, come segue. La lista b riutilizza tutti gli elementi di a, la lista c riutilizza l'elemento

20 di a e b, mentre la lista d riutilizza la sola costante Nil() che indica la lista vuota al fondo di a,b,c. Ecco il disegno:

Le liste a,b,c,d di cui sopra si sovrappongono in parte



Ecco la definizione di List e delle sue sottoclassi.

```
// List.java
// Liste crescenti di interi per rappresentare insiemi
// INVARIANTE della classe: ogni lista in List e' crescente
// La classe astratta "Lista" elenca i metodi che voglio
// definire.
// Le sottoclassi Nil, Cons realizzano questi metodi nei vari casi:
// Nil:  nel caso di una lista con zero elementi (vuota)
// Cons: nel caso di una lista con almeno un elemento

public abstract class List
{// Zero costruttori per List. Tutti i metodi di List sono astratti.
 // Elenco dei metodi astratti di List da sovrascrivere
 public abstract boolean empty(); //controllo se this=vuota
```

```
 public abstract int size();      //numero elementi di this
 public abstract boolean contains(int x); //controllo se x in this
 public abstract List insert(int x);  //nuova lista=this unione {x}
 public abstract List append(List l); //nuova lista=this unione l

 //Tutti i metodi di tipo List costruiscono una nuova lista,
 //senza modificare this e la lista in input
 //In Nil e Cons sovrascrivo anche il metodo "toString"
 //per descrivere una lista con una stringa.
}



//Nil.java LISTE VUOTE
//Sottoclasse concreta (= non astratta) di List:
//sovrascriviamo tutti i metodi astratti di List.
//L'unico elemento di Nil e' null e rappresenta la lista vuota.

/* NOTA. Questa classe non si puo' compilare prima di aver
sovrascritto tutti i metodi astratti di List, ne' prima della classe
Cons, perche' usa il il costruttore di Cons */

public class Nil extends List
{ /* Costruttore di default new Nil() Riscriviamo i metodi astratti
di List e il metodo toString nel caso della lista vuota */

public boolean empty(){ return true; }
/* empty() e' costante = true sulla sottoclasse Nil, e' costante =
false sulla sottoclasse Cons, dunque NON e' costante su List */

public int size(){ return 0; }
/* size() e' costante = 0 sulla sottoclasse Cons, NON e'
costante sulla sottoclasse Cons, dunque NON e' costante su List */

public boolean contains(int x){ return false; }
/* contains(x) e' costante = false sulla sottoclasse Nil, NON e'
costante sulla sottoclasse Cons, dunque NON e' costante su List */

//INSERT. Metodo che aggiunge x a una lista
//Quando aggiungo un elemento costruisco una nuova lista non vuota,
//dunque nella sottoclasse concreta Cons. Devo quindi usare new e il
//costruttore Cons per descrivere il risultato, e non posso compilare
```

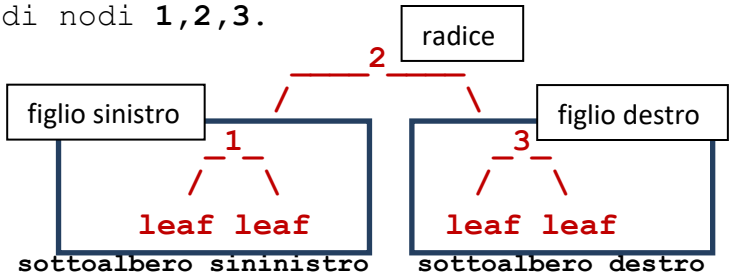
```
//Nil prima di compilare Cons.
public List insert(int x){ return new Cons(x, this); }
/* NOTA: qui this indica l'unico elemento della classe Nil, la lista
vuota. Evitando di scrivere new Nil() riutilizzo la precedente
versione della lista vuota. */

/* TOSTRING. Sovrascrivo il metodo toString() (che fa parte di ogni
classe) per definire una stringa (vuota) che rappresenta la lista
vuota */
public String toString(){ return ""; }

public List append(List l){ return l; }
/* L'unione di una lista vuota e di l e' l. Restituendo l stessa
evito di definire un clone di l e risparmio memoria. */
}



//Cons.java LISTE NON VUOTE

//Sottoclasse concreta (= non astratta) di List:
//sovrascriviamo tutti i metodi astratti di List.
//Gli elementi di Cons rappresentano le liste NON vuote.
//Le implementazioni sono ricorsive

public class Cons extends List
{ //Una lista (ordinata) non vuota ha due informazioni:
  private int elem;  //primo elemento
  private List next; //indirizzo degli elementi rimanenti

/* Definisco il costruttore Cons come protected, perche' consente di
costruire liste non ordinate, mentre vogliamo impedire a chi usa la
classe di farlo. Usando protected possiamo usare Cons in Nil. */
protected Cons(int elem, List next)
{this.elem = elem; this.next = next;}

// Riscriviamo i metodi astratti di List e il metodo toString
// nel caso della lista NON vuota. Quando restituiamo una lista
// vogliamo avere o uno degli argomenti del metodo oppure una
// lista nuova, quindi usiamo new e il costruttore Cons

public boolean empty(){ return false; }
```

```
// empty() e' costante = false sulla sottoclasse Cons, e' costante
// = true sulla sottoclasse Nil, dunque NON e' costante su List

public int size(){ return 1 + next.size(); }
// size() chiama ricorsivamente se stesso se next e' in Cons,
// chiama il metodo size() di Nil se next e' in Nil

public boolean contains(int x)
{ return x == elem || next.contains(x); }
//Il metodo contains(x) chiama ricorsivamente se stesso se next e'
//in Cons, chiama il metodo contains(x) di Nil se next e' in Nil

//INSERT. Metodo che aggiunge x, costruisce una nuova lista
//riutilizzando this se possibile, e preserva l'ordine crescente
public List insert(int x){
//Se x piu' piccolo del primo elemento aggiungo x davanti a tutti
 if (x < elem)
     return new Cons(x, this);
//Se x uguale al primo elemento lascio this come si trova
 else if (x == elem)
     return this;
//Se x maggiore del primo elemento aggiungo x nel resto della lista
 else //in questo caso x>elem
     return new Cons(elem, next.insert(x));}
//Il metodo insert(x) chiama ricorsivamente se stesso se next e'
//in Cons,  chiama il metodo insert(x) su Nil se next e' in Nil


//APPEND. Aggiunge una lista l a this, costruendo una nuova
//lista e preservando l'ordine crescente. Usiamo insert per
//aggiungere il contenuto di l un elemento alla volta
//preservando l'ordine ad ogni passo
 public List append(List l)
 {if (l.empty())
   return this;
  else //prima aggiungo il primo elemento di l, dopo gli altri
  {int x = ((Cons) l).elem;  //x = primo elemento di l
   List m = ((Cons) l).next; //m = altri elementi di l
   return insert(x).append(m);}}

/* DOWNCAST: per scrivere l.elem, l.next devo prima fare un downcast
((Cons) l) per spostare l in Cons, dato che elem, next esistono solo
```

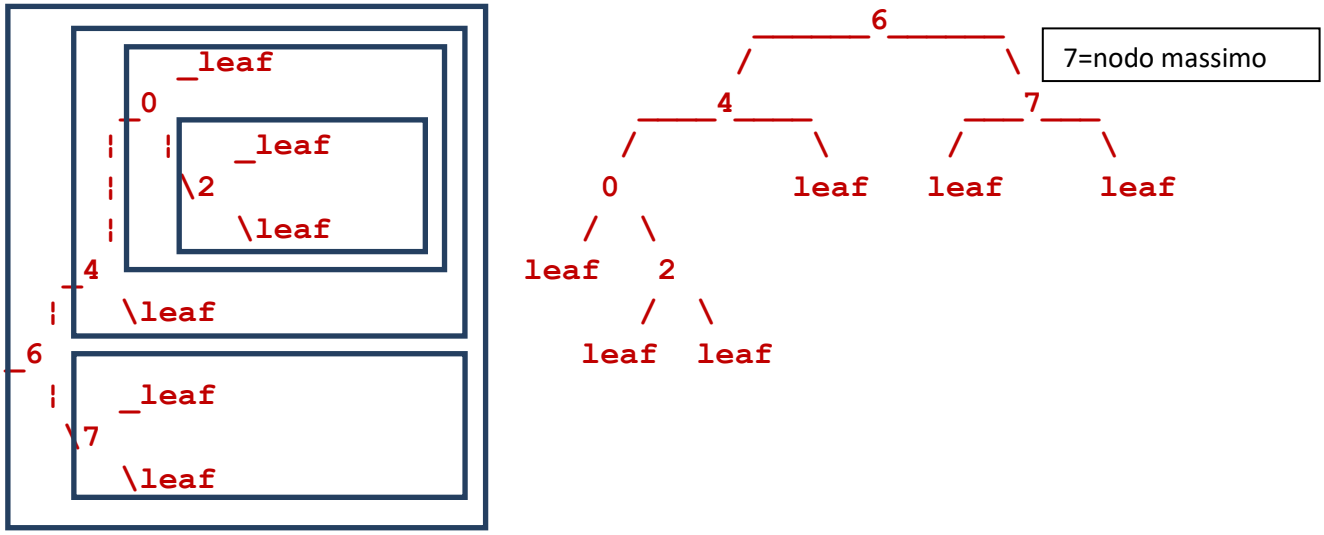
```
in Cons. Il downcast non causa errori al run-time perche' se l non e'
vuota ha tipo esatto Cons. */

//TOSTRING: metodo che restituisce una stringa che descrive la lista
public String toString() // trasformo il primo elemento poi gli altri
 {return elem + " " + next.toString();}}



//TestList.java
//Proviamo a costruire e stampare delle liste realizzate con la
//classe astratta List

public class TestList
{public static void main(String[] args)
{List a = new Nil(); a=a.insert(10).insert(20);
 List b = a.insert(5); List c = b.insert(15); List d = c.insert(25);
 System.out.println( " d = " + d);

 System.out.println( "\nEsempi di unione di l={1,2} con m={3,4}");
 System.out.println( "Aggiungendo l e m in ordine diverso il
risultato non cambia");
 List l = new Nil(); l=l.insert(1).insert(2);
 List m = new Nil(); m=m.insert(3).insert(4);
 System.out.println( " l.append(m) = " + l.append(m));
 System.out.println( " m.append(l) = " + m.append(l));
 System.out.println( "Aggiungere due volte non cambia il risultato");
 System.out.println( " m.append(l).append(l) = " +
m.append(l).append(l)); }}
```

Lezione 17 La classe astratta alberi di ricerca

Lezione 17. La definizione ricorsiva di albero binario. Gli alberi binari di interi sono ***alberi vuoti***, indicati con ***"leaf"***, oppure hanno una radice, costituita da un intero, e un sottoalbero sinistro e uno destro. Le radici dei sottoalberi sono dette i nodi dell'albero, le radici del sottoalbero sinistro e destro (quando esistono) sono dette i figli sinistri e destri. Un albero binario è di ricerca se la radice è maggiore di ogni nodi nel sottoalbero sinistro, e minore di ogni nodo nel sottoalbero destro, e lo stesso vale per ogni nodo dell'albero. In particolare ***non ci sono nodi ripetuti***. Come esempio, ecco un albero di nodi **1,2,3.**



Si tratta di un albero di ricerca perché l'unico nodo che si trova a sinistra, 1, è minore della radice 2, che è minore dell'unico nodo 3 a destra. Tutto ciò che conta in un albero sono i collegamenti tra i nodi. Ecco due modi di disegnare lo stesso albero, con nodi 0,2,4,6,7, nel primo disegno i sottoalberi sono disposti dall'alto in basso, nel secondo da sinistra a destra. Anche in questo caso si tratta di un albero di ricerca. Nel primo disegno, i sottoalberi non vuoti sono evidenziati con dei rettangoli.

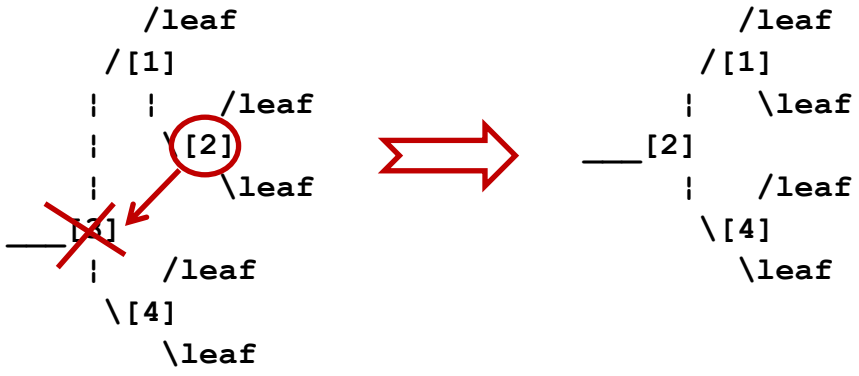


Rappresentazione grafica dell'albero con i suoi rami e il nodo massimo che è uguale a 7

Possiamo definire gli alberi di ricerca per ogni tipo di dati per cui abbiamo definito un ordine (totale). In questa lezione ci occupiamo di definire una classe astratta Tree per ***gli alberi di ricerca di interi***: la costruzione è simile a quella delle liste astratte viste nella Lezione 17. La struttura Tree ha la caratteristica di essere dinamica (cambia di dimensione a seconda della necessità), e di poter effettuare ricerca, inserimento e cancellazione di un nodo in tempo proporzionale al logaritmo del numero dei nodi, quindi molto in fretta. Tuttavia, nel corso di ProgII non ci occupiamo di questo secondo aspetto, che sarà trattato nel corso di Algoritmi.

Come metodi della classe Tree definiamo ***boolean empty()*** per decidere se un albero è vuoto, ***int max()*** per calcolare il nodo massimo di un albero, ***boolean contains(int x)*** per decidere se un albero contiene il nodo x, e infine ***Tree insert(int x)*** per e ***Tree remove(int x)*** oer aggiungere/rimuovere un nodo x. Le modifiche devono preservare il fatto che ogni albero è un albero di ricerca.

Nota sulla rimozione di un nodo.



Non è semplice ***rimuovere*** un nodo x da un albero di ricerca e ottenere ancora un albero di ricerca. Consideriamo il caso in cui ***il nodo da eliminare è la radice e entrambi i sottoalberi non sono vuoti***. Prendiamo l'albero di nodi 1,2,3,4 qui sopra: se eliminiamo la radice 3, scomponiamo l'albero nel sottoalbero sinistro, di nodi 1,2, e nel sottoalbero destro con il solo nodo 4. Dobbiamo connettere i due alberi, con una radice ***scelta in modo tale da ottenere un albero di ricerca***. A tal fine, rimpiazzeremo il nodo 3 con il massimo nodo del sottoalbero sinistro (il nodo 2 in questo caso). In questo modo evitiamo di disconnettere l'albero e inseriamo una radice maggiore o uguale di ogni nodo a sinistra e minore di ogni nodo a destra. ***Per***

avere un albero di ricerca, dobbiamo ancora evitare di ripetere il nodo 2 nella radice e nel sottoalbero sinistro. È sufficiente eliminare il nodo 2 dal sottoalbero sinistro: dato il sottoalbero sinistro è un albero più piccolo, questa è una definizione corretta di un algoritmo ricorsivo per rimuovere un nodo da un albero.

```
//Tree.java         classe astratta alberi binari di ricerca
// INVARIANTE DI CLASSE: ogni oggetto e' un albero di ricerca
// Realizziamo gli alberi binari con 2 sottoclassi concrete:
// LEAF: contiene il solo albero vuoto "leaf"
// BRANCH: contiene tutti gli alberi non vuoti

// La classe astratta contiene il nome della classe, "Tree", e il
// minimo di metodi che richiediamo per formare una classe concreta
// di alberi.
// In questa implementazione quando inseriamo cancelliamo: dunque
// il passato di un albero NON e' ricostruibile

public abstract class Tree {
//Test se l'albero e' vuoto
public abstract boolean empty();

//Massimo elemento dell'albero, se non vuoto:
//in un albero binario e' il nodo piu' a destra
public abstract int max();

//Test di appartenenza
public abstract boolean contains(int x);

// Aggiunta di un elemento a un albero. Modifica l'albero precedente,
// la cui forma originaria va persa.
public abstract Tree insert(int x);
// Si usa con t = t.insert(x), per salvare le modifiche fatte a t

// Rimozione di un elemento da un albero (se c'e'). Modifica l'albero
// precedente, che va perso.
public abstract Tree remove(int x);
// Si usa con t = t.remove(x), per salvare le modifiche fatte a t

protected abstract void scriviOutput
```

```
  (String prefix, String root, String left, String right);
//Metodo che gestisce la parte NON pubblica della stampa.
//Non forniamo spiegazioni sul suo funzionamento.

 public void scriviOutput()
  {scriviOutput("","___","   ","   ");}
// Metodo pubblico di stampa, dall'alto verso il basso, con i
// sottoalberi disegnati piu' a destra dell'albero di cui fan parte
}

//Leaf.java     unico elemento: "leaf" (albero vuoto)

// Implementazione della classe Leaf per rappresentare alberi vuoti
// I metodi definiti in Leaf restituiscono risultati costanti
// (che non dipendono dall'albero).

public class Leaf extends Tree {
  public Leaf() { }
// Il costruttore non assegna nulla e si  puo' lasciare implicito.
// l'albero vuoto "leaf" e' l'unico oggetto della classe. Un albero
// viene inizializzato da new Leaf() e esteso un elemento alla volta

public boolean empty(){ return true; } //l'albero vuoto e' vuoto

public int max(){assert false; return 0;} /* l'albero vuoto non ha
massimo, e' sbagliato chiederlo. Java richiede un return se c'e' un
tipo di ritorno, per questo scriviamo return 0;   */

public boolean contains(int x) { return false; }
//l'albero vuoto non contiene nulla

public Tree insert(int x) { return new Branch(x, this, this);}
//se inserisco x ottengo l'albero di radice x e nessun figlio
//per rappresentare "Leaf()" senza creare foglie nuove scrivo "this":
//nella classe "Leaf" abbiamo sempre: this=new Leaf()

public Tree remove(int x) { return this; } //non c'e' nulla da
// cancellare nell'albero vuoto, quindi non cambia nulla


//Metodo che gestisce la parte NON pubblica della stampa.
//Non forniamo spiegazioni sul suo funzionamento.
```

```
 protected void scriviOutput
 (String prefix, String root, String left, String right)
 { System.out.println(prefix + root + "leaf"); }}


/* Branch.java  Sotto-classe di Tree degli alberi non vuoti:
                                elem
                              /      \
                           left      right
 Gli elementi a sinistra sono minori di elem, quelli a destra sono
 maggiori */

public class Branch extends Tree {
 private int elem;   //radice
 private Tree left;  //nodi a sinistra: piu' piccoli della radice
 private Tree right; //nodi a destra: piu' grandi della radice

 public Branch(int elem, Tree left, Tree right)
 {this.elem = elem; this.left = left; this.right = right;}

public boolean empty(){ return false; }
// Un albero non vuoto non e' vuoto


public int max(){ return right.empty() ? elem : right.max(); }
// se la parte destra e' vuota il nodo piu' grande e' la radice.
// Altrimenti il nodo piu' grande si trova a destra

public boolean contains(int x) /* Usa la RICERCA BINARIA, in media
richiede tempo log_2(n) dove n = numero dei nodi. */
{if (x == elem) // abbiamo trovato l'elemento
  return true;
 else if (x < elem)
// x se c'e' si trova tra i nodi piu' piccoli a sinistra
  return left.contains(x);
 else //x>elem
// x se c'e' si trova tra i nodi piu' grandi a destra
 return right.contains(x);}

public Tree insert(int x) /* Inseriamo x preservando l'invariante
"albero di ricerca": dunque x va inserito a sinistra se e' piu'
piccolo della radice e a destra se e' piu' grande */
```

```
{if (x < elem)
  left = left.insert(x); //e' essenziale aggiornare il valore di left
 else if (x > elem)
  right = right.insert(x); //e' essenziale aggiornare il valore right
// altrimenti x=elem, x gia' presente nell'albero, non lo inseriamo

 return this; } /* Devo ricordarmi di restituire il valore aggiornato
dell'albero, altrimenti la modifica fatta puo' andare persa */

public Tree remove(int x){
 if (x == elem) // trovato elemento da eliminare
     if (left.empty())
  // il sottoalbero sinistro e` vuoto, dunque resta il
  // sottoalbero destro
  return right;
     else if (right.empty())
  // il sottoalbero destro e` vuoto, dunque resta il
  // sottoalbero sinistro
  return left;
     else {elem = left.max(); //rimpiazziamo elem con il massimo sin.
// per evitare ripetizioni,eliminiamo il massimo dal sottoalbero sin.
 left = left.remove(elem); //aggiorno il valore di left
 return this;}
 else if (x < elem) {
 // se c'e', l'elemento da eliminare e` nel sottoalbero sinistro
 left = left.remove(x); //aggiorno left
 return this;}
 else {
 // se c'e', l'elemento da eliminare e' nel sottoalbero destro
  right = right.remove(x); //aggiorno right
  return this;}}

//Metodo che gestisce la parte NON pubblica della stampa.
//Non forniamo spiegazioni sul suo funzionamento.
protected void scriviOutput
(String prefix, String root, String left, String right)
{this.left.scriviOutput(prefix+left, "   /", "   ", "  ¦");
 System.out.println(prefix + root + "[" + elem + "]");
 this.right.scriviOutput(prefix+right, "   \\", "  ¦", "   ");}}

//TestTree.java    sperimento la classe Tree degli alberi di ricerca
import java.util.*;
```

```
//Inserisco la libreria di utilities Java, per avere la classe Random
public class TestTree {public static void main(String[] args){
 Random r = new Random(); //r e' un generatore di numeri casuali

// Creo un albero t con n numeri interi casuali tra 0 e (n-1)
// (gli interi estratti piu' volte compaiono una volta sola, altri
// interi tra 0 e (n-1) non compaiono affatto)
 int n = 8;
 Tree t = new Leaf(); //L'albero t nasce vuoto
 for (int i = 0; i < n; i++)
  t = t.insert(r.nextInt(n)); //Accresco t un elemento alla volta

//Provo il metodo di stampa e il calcolo del massimo
 System.out.println( "Stampa albero casuale t di al piu' " + n + "
elementi"); t.scriviOutput(1);
 System.out.println( "\n t.max() = " + t.max());

//Creo un albero u inserendo sempre elementi piu' grandi
//quindi sempre a destra
 Tree u = new Leaf();
 for (int i = 0; i < n; i++) u = u.insert(i);
 System.out.println( "Stampa albero u di " + n + " elementi, tutti
figli destri"); u.scriviOutput(1);

//Creo un albero u inserendo sempre elementi piu' piccoli
//quindi sempre a sinistra
 Tree v = new Leaf();
 for (int i = n-1; i >=0; i--)  v = v.insert(i);
 System.out.println( "Stampa albero v di " + n + " elementi,
elementi, tutti figli sinistri"); v.scriviOutput(1);

 Tree w = new Leaf (); w=w.insert(3); w=w.insert(1); w=w.insert(4);
 w=w.insert(2);
 System.out.println( "Stampa albero w con insieme nodi={1,2,3,4}");
 w.scriviOutput();
 System.out.println( "w senza il nodo 3");
 w.remove(3);w.scriviOutput();

}}
```

Lezione 18 Interfacce, generici vincolati e alberi di ricerca

Lezione 18. In questa lezione usiamo i generici vincolati per definire una classe astratta degli alberi su una generica classe T.

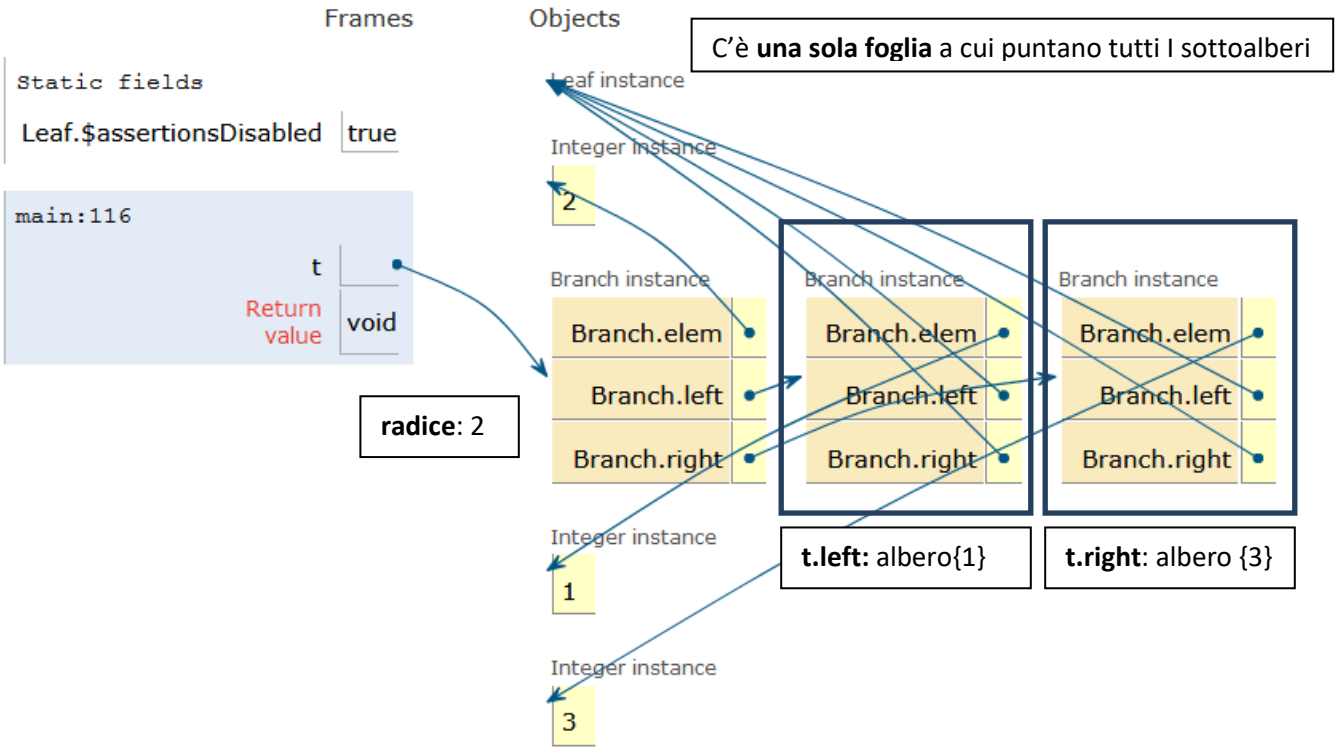
Interfaccie Java. Abbiamo prima bisogno della nozione di interfaccia. Una interfaccia è una classe astratta con solo metodi astratti: si presenta come una lista di segnature di metodi, senza nessuna implementazione. Una interfaccia viene definita usando la parola chiave "interfaccia", al posto di "classe astratta". Per estendere una interfaccia, al posto della parola chiave "extends", usiamo la parola chiave "implements". Per implementare una interfaccia dobbiamo sovrascrivere con metodi concreti tutti i suoi metodi, come già visto per le classi astratte. Una classe Java può estendere al più una classe, ma può implementare un numero qualunque di interfaccie. Il motivo è che due classi astratte A, B possono fornire due versioni concrete diverse dello stesso metodo m, e se vogliamo che una classe C estenda A, B abbiamo un conflitto su quale versione di m ereditare. Questo non succede aggiungendo un numero qualsiasi di interfaccie, che non forniscono mai versioni concrete di un metodo.

L'interfaccia Comparable<T>. Come esempio, consideriamo l'interfaccia predefinita ***Comparable<T>***, con un unico metodo (astratto) ***int compareTo(T y)***. Questo metodo confronta due elementi di T: x.compareTo(y) è negativo se x precede y, è 0 se x e y sono uguali, è positivo se x è maggiore di y. Quando dichiariamo che C implementa Comparable<C>, siamo tenuti a fornire in C una implementazione per ***int compareTo(C y)***, e così facendo scegliamo una nozione di eguaglianza e una nozione di ordine per C. Come esempio: le classi predefinite ***Integer***, ***Double***, ***String*** implementano Comparable<T> rispettivamente per T=Integer, Double, String.

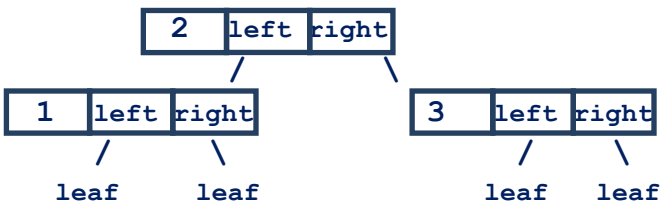
Generici Vincolati. Possiamo definire una classe astratta Tree<T> di alberi di ricerca su T aggiungendo un vincolo alla classe generica T che utilizziamo: possiamo scrivere ***Tree<T extends I>***, dove I è una classe astratta oppure una interfaccia. In questo caso quando descriviamo Tree<T> abbiamo a disposizione tutti i metodi di I. Però in questo modo abbiamo contratto un debito, e quando vogliamo utilizzare Tree<C> per una particolare classe C possiamo solo scegliere una classe che estende I, altrimenti Java segnala un errore. Quando scriviamo ***Tree<T extends I>***, diciamo che T è un generico vincolato.

La classe astratta Tree<T extends Comparable<T>> degli alberi di ricerca sulla classe T. Per costruire la classe astratta degli alberi di ricerca su una classe T generica abbiamo bisogno di un vincolo su T: chiediamo che T estenda Comparable<T>. In questo modo possiamo usare compareTo per confrontare tra loro gli oggetti di T e definire gli alberi di ricerca su T. Quando scegliamo un valore per T, dobbiamo scegliere una classe C che estenda Comparable<C>, cioè che fornisca una implementazione per ***int compareTo(C y)***.

Un esempio. L'albero di ricerca t={1,2,3}, di radice 2 e figli 1 e 3, il cui tipo è ottenuto istanziando Tree<T> su T=Integer.



Ecco invece il solo albero t disegnato con t.left a sinistra e t.right a destra, considerando le foglie distinte tra loro e gli interi 1,2,3 parte dell'albero.



//Tree.Java

```
public abstract class Tree<T extends Comparable<T>>
{ //alberi di ricerca su T
  public abstract boolean empty();
  public abstract boolean contains(T x);
  public abstract T max();
  public abstract Tree<T> insert(T x);
  public abstract Tree<T> remove(T x);

  protected abstract void scriviOutput
    (String prefix, String root, String left, String right);

public void scriviOutput()
  {scriviOutput("","___","    ","    ");}}

public class Leaf<T extends Comparable<T>>
  extends Tree<T>
{ //ALBERI VUOTI" unico elemento "leaf"
  //non definisco nessun costruttore: di default ho new Leaf()

  public  boolean empty(){return true;}
  public  boolean contains(T x){return false;  }
  public  T max(){assert false; return null;    }
  public  Tree<T> insert(T x){return new Branch<T>(x, this, this);}
  public  Tree<T> remove(T x){return this;}

  protected void scriviOutput
      (String prefix, String root, String left, String right)
        { System.out.println(prefix + root + "leaf"); }



}

public class Branch<T extends Comparable<T>> extends Tree<T>
{ //alberi non vuoti
  private T elem; Tree<T> left; Tree<T> right;
  public Branch(T elem, Tree<T> left, Tree<T> right)
  {this.elem=elem; this.left=left; this.right=right;}
  public  boolean empty(){return false;}

  public  boolean contains(T x)
  {int c=x.compareTo(elem);
```

```
    if (c==0) //x=elem
     return true;
    else if (c<0) //x<elem
      return  left.contains(x);
    else //c>0, x>elem
      return right.contains(x);}

  public  T max()
  {if (right.empty())
     return elem;
    else //right non vuoto
      return right.max();
  }

  public  Tree<T> insert(T x)
    { int c=x.compareTo(elem);
       if (c<0) //x<elem
       {left=left.insert(x);}
      else if (c>0) //x>elem
      {right=right.insert(x);}
      //se c=0 allora x=elem e non inserisco x
      return this;}

  public  Tree<T> remove(T x)
    { int c=x.compareTo(elem);
       if (c<0) //x<elem
       {left=left.remove(x); return this;}
       else if (c>0) //x>elem
       {right=right.remove(x); return this;}
      else /* x=elem */ if (left.empty())
        return right; //cancello elem, se left=leaf resta right
      else if (right.empty())
        return left; //cancello elem, se right=leaf resta left
      else ////cancello elem, left e right non sono vuoti
       {elem=left.max(); //rimpiazzo elem con il massimo a sinistra
        left.remove(elem); //per evitare ripetizioni
        return this;}}

//Metodo che gestisce la parte NON pubblica della stampa.
//Non forniamo spiegazioni sul suo funzionamento.
protected void scriviOutput
(String prefix, String root, String left, String right)
```

```
{this.left.scriviOutput(prefix+left, "   /", "    ", "  ¦");
 System.out.println(prefix+root +"["+ elem +"]");
 this.right.scriviOutput(prefix+right, "   \\", "  ¦", "    ");}}

//Contatto.java Forniamo una classe C che implementa Comparable<C>
public class Contatto implements Comparable<Contatto>
{
  private String nome; private String email;
  public Contatto(String nome, Stringemail)
   {this.nome=nome;this.email=email;}
  public String getNome(){return nome;}
  public void setNome(String nome){this.nome=nome;}
  public String getEmail(){return email;}
  public void setEmail(String nome){this.email=email;}
  public int compareTo(Contatto x){return nome.compareTo(x.nome);}
  public String toString(){return nome + "<" + email + ">";}
  public Contatto key(String nome){return new Contatto(nome, null);}
}


//TestTree.java
import java.util.*; //Per la classe Random
// Provo l'implementazione degli alberi binari come classe astratta

public class TestTree {
  public static void Title(String s)
  {System.out.println( "--------------------------------------");
    System.out.println(s);
    System.out.println( "--------------------------------------");}

    public static void main(String[] args){
 Random r = new Random(); //r = un generatore di numeri casuali
 // Creo un albero t con n reali casuali tra 0 e 1
 int n = 8;
 Tree<Double> t = new Leaf<Double>(); //L'albero t nasce vuoto
 for (int i = 0; i < n; i++)
     t = t.insert(r.nextDouble()); //Accresco t un elem. alla volta

//Provo il metodo di stampa e il calcolo del massimo
 Title( "Stampa albero casuale t di " + n + " elementi");
 t.scriviOutput();
 System.out.println( "\n t.max() = " + t.max());
```

```
//Creo un albero u inserendo sempre elementi piu' grandi
//quindi sempre a destra
 Tree<Integer> u = new Leaf<Integer>();
 for (int i = 0; i < n; i++) u = u.insert(i);
 Title( "Stampa albero u di " + n + " elementi, tutti figli destri");
 u.scriviOutput();

//Creo un albero u inserendo sempre elementi piu' piccoli
//quindi sempre a sinistra
 Tree<String> v = new Leaf<String>();
 for (int i = n-1; i >=0; i--) v = v.insert( "numero " + i);
 Title("Stampa albero v di "+n+" elementi, tutti figli sinistri");
 v.scriviOutput();

//Provo il metodo di cancellazione
 Tree<Contatto> w = new Leaf<Contatto>();
 Contatto c = new Contatto( "Cafasso", "cafasso@ristorante"));
 Contatto a = new Contatto( "Anfossi", "anfossi@scuola"));
 Contatto d = new Contatto( "Davanzo", "davanzo@comune"));
 Contatto b = new Contatto( "Borghi",  "borghi@ditta"));
 w=w.insert(c);w=w.insert(a);w=w.insert(d);w=w.insert(b);

 Title("Stampa albero w di contatti {a,b,c,d}"); w.scriviOutput();
 Title("w senza il contatto c"); w.remove(c);w.scriviOutput();}}
```

Lezione 19 Interfacce Comparable, Iterator e iterable

Lezione 19. Parte 1. La classe Bottiglia come implementazione dell'interfaccia Comparable<Bottiglia>.

35 minuti. Se dichiariamo che Bottiglia implementa l'interfaccia Comparable<Bottiglia> dobbiamo aggiungere alla classe Bottiglia una implementazione del metodo astratto ***int compareTo(Bottiglia b)***, per confrontare due bottiglie. In cambio, possiamo usare i metodi generici ***void Arrays.sort()*** e ***void Arrays.binarysearch()*** per ordinare un vettore di bottiglie e per la ricerca ordinata in un vettore ordinato di bottiglie. Questi metodi possono essere applicati a vettori su una classe T generica, ma solo ***se T implementa Comparable<T>***.

Ricordiamo che il metodo ***int compareTo(Bottiglia b)*** deve restituire 0 se l'oggetto this e` uguale all'oggetto b passato come parametro, un valore > 0 se l'oggetto this è piu' grande di quello passato come parametro, un valore < 0 altrimenti. Chi progetta la classe decide come fare il confronto e di conseguenza l'implementazione del metodo. In questo caso, decidiamo di confrontare sue bottiglie solo in base al loro livello, ignorandone la capacità. Consideriamo equivalenti due bottiglie con lo stesso livello, anche se una è una damigiana e l'altra una bottiglietta.

Un dettaglio: in questa lezione utilizziamo il costrutto "foreach" su vettori, definito da for(C c:v){…c…} per v vettore di elementi della classe C. Il costrutto esegue {…c…} per c=v[0], …, v[n-1] in quest'ordine, con n=v.length().

```
//Bottiglia.java Riprendiamo la classe Bottiglia della Lezione 5
public class Bottiglia implements Comparable<Bottiglia>{
private int capacita, livello;  // 0 <= livello <= capacita'
 public Bottiglia(int capacita)
 {this.capacita = capacita;
  livello = 0;
  assert (0<=livello) && (livello <= capacita);}

// Restituiamo la quantita' effettivamente aggiunta
public int aggiungi(int quantita)
 {assert quantita >= 0:
  "la quantita' doveva essere >=0 invece vale " + quantita;
  int aggiunta = Math.min(quantita, capacita-livello);
```

```
  livello = livello + aggiunta;
  assert (0<=livello) && (livello <= capacita);
  return aggiunta;}


// Restituiamo la quantita' effettivamente rimossa
 public int rimuovi(int quantita)
 { assert quantita >= 0:
  "la quantita' doveva essere >=0 invece vale " + quantita;
  int rimossa = Math.min(quantita, livello);
  livello = livello - rimossa;
  assert (0<=livello) && (livello <= capacita);
  return rimossa;}

 public int getCapacita(){ return this.capacita; }
 public int getLivello() { return this.livello; }
// Non Consentiamo di cambiare la capacita'

 public void setLivello(int livello)
 {this.livello = livello;
  assert (0<=livello) && (livello <= capacita);}

 public String toString()
 {return livello + "/" + capacita;}

public int compareTo(Bottiglia b){return   livello - b .livello;}
// La differenza di livello e' 0 se le bottiglie hanno lo stesso
// livello, e' >0 se this ha livello maggiore, e' <0 altrimenti
}


import java.util.*; //Per la classe Arrays
// COSA ABBIAMO FATTO: nella classe Bottiglia abbiamo sovrascritto
// il metodo compareTo, per confrontare due bottiglie
// COSA OTTENIAMO IN CAMBIO: i metodi statici Arrays.sort()
// e Arrays.binarysearch() per ordinare un vettore di bottiglie e
// per la ricerca binaria in un vettore ordinato di bottiglie

public class ComparaBottiglie {public static void main(String[] args)
{Bottiglia b1 = new Bottiglia(10); //bottiglia vuota di capacita' 10
 Bottiglia b2 = new Bottiglia(20); //bottiglia vuota di capacita' 20
 Bottiglia b3 = new Bottiglia(5);  //bottiglia vuota di capacita'  5
```

```
 Bottiglia b4 = new Bottiglia(15); //bottiglia vuota di capacita' 15
//Riempo le prime 3 bottiglie, poi le confronto in base al livello
 b1.aggiungi(3); // b1 e' la piu' piena (la capacita' e' irrilevante)
 b2.aggiungi(2); // b2 e' intermedia
 b3.aggiungi(1); // b3 e' la meno piena
//b4 resta vuota

//confronto in base al livello
 System.out.println( " confronto  b1 (3 litri) con b2 (2 litri): " +
b1.compareTo(b2));
 System.out.println( " confronto  opposto: " +  b2.compareTo(b1));
 System.out.println( " confronto b1 con b1: " + b1.compareTo(b1));

//ordinamento di un vettore di bottiglie in base al livello
 Bottiglia[] bottiglie = {b1, b2, b3}; //non aggiungo b4
 // posso ordinare le bottiglie in questo array perche'
 // Bottiglia implementa l'interfaccia Comparable

 System.out.println(" Ordino e stampo {b1, b2, b3}");
 Arrays.sort(bottiglie);
//Dato che "bottiglie" e' un vettore posso usare il "foreach"
 for (Bottiglia b : bottiglie) System.out.println(b);

//posso eseguire la ricerca binaria della posizione di una bottiglia
//in questo array ordinato perche' Bottiglia implementa l'interfaccia
//Comparable<Bottiglia>. binarySearch(b) restituisce un numero
//negativo se b non e' presente

 System.out.println( " cerco posizione di b1 (3 litri) nell'array: "
+ Arrays.binarySearch(bottiglie, b1));
 System.out.println( " cerco posizione di b2 (2 litri) nell'array: "
+ Arrays.binarySearch(bottiglie, b2));
 System.out.println( " cerco posizione di b3 (1 litro) nell'array: "
+ Arrays.binarySearch(bottiglie, b3));
 System.out.println( " cerco posizione di b4 (0 litri) nell'array: "
+ Arrays.binarySearch(bottiglie, b4));}}
```

Lezione 19. Parte 2. Le interfaccie Comparable<T>, Iterable<E> e Iterator<E>. Una classe T che implementa due interfaccie. *65 minuti.*

L'interfaccia Iterable<E>. Spieghiamo l'uso di Iterable<E>. Supponiamo di voler scrivere un for che passi attraverso una lista l di elementi di tipo E, ma ***senza rendere pubblici gli indirizzi dei nodi all'interno di l***. Vogliamo impedire di alterare i puntatori tra nodi di l. La soluzione è usare una classe T di liste di oggetti di tipo E che implementa Iterable<E>: l'implementazione di Iterable<E>, a sua volta, richiede ua classe che implementa l'interfaccia Iterator<E>. Spieghiamo come.

Per implementare Iterable<E>, dobbiamo definire in T un metodo **public L iterator()** che rietichetta una lista l di T come un oggetto di L, dove L è una classe che implementa una ***seconda interfaccia, Iterator<E>***. Iterator<E>, a sua volta, richiede vengano implementati il metodo ***boolean hasNext()***, per sapere quando gli oggetti di l sono finiti, e il metodo ***E Next()***, per conoscere il valore dell'elemento e:E nel nodo corrente di l, e per spostarsi al nodo successivo. Una volta implementati Iterable<E> e Iterator<E>, possiamo usare in T usare il costrutto *foreach*: ***for (E e:l){…e…}***, dove l ha tipo T. Un foreach esegue {…e…} un volta per ogni elemento e della lista l, nell'ordine, senza rendere pubblici gli indirizzi dei nodi di l.

Abbiamo bisogno di importare pacchetto java.util dove risiedono le dichiarazioni delle interfaccie Iterator e Iterable, per poter dichiarare che la classe T di liste su E implementa l'interfaccia Iterable<E>.

Definiamo come esempio una classe ***ListExt*** di liste su interi che implementa ***Comparable<ListExt>*** e ***Iterable<Integer>***, e una classe ***ListIterator*** che implementa ***IteratorInteger>.*** Notate che dobbiamo scrivere ***Integer*** per il tipo degli elementi di ListExt e non ***int***. Il motivo è che gli interi Java non sono una classe, ma un generico deve essere una classe. Quindi usiamo la rappresentazione degli interi come classe, la classe Integer. Come abbiamo già detto, Integer è un semplice "wrapper" di int, cioè gli oggetti di Integer sono gli oggetti il cui unico attributo è un intero.

Scegliamo di implementare due interfaccie in ***ListExt***: la seconda è Comparable<ListExt> che confronta due liste in ListExt. Come implementazione di compareTo per liste di interi scegliamo il

confronto ***lessicografico*** tra liste, analogo all'ordine tra le parole del vocabolario. Una lista è minore di un'altra se la prima è prefisso (parte iniziale) della seconda, oppure se nella prima posizione in cui le due liste sono diverse la prima ha un elemento minore della seconda.

```
// Node.java   Riutilizziamo la classe Node della Lezione 8.
// un nodo contiene un valore e un riferimento al nodo
// successivo (null se non ce ne sono)
public class Node {private int elem; private Node next;
 public Node(int elem, Node next){this.elem = elem;this.next = next;}
 public void setElem(int elem){this.elem = elem;}
 public int getElem(){return elem;}
 public Node getNext(){return next;}
 public void setNext(Node node){next = node;}}

// ListExt.java modifichiamo MiniLinkedList della Lezione 14
// cambiando il nome della classe e aggiungendo le implementazioni
// richieste.

import java.util.*; //per le interfaccie
// Dichiariamo che ListExt implementa le interfacce Iterable e
// Comparable. Questo consentira` da un lato di usare il costrutto
// iterativo for-each di Java per iterare sugli elementi di una
// struttura e dall'altro di confrontare istanze della classe
// secondo l'ordine lessicografico

public class ListExt implements Iterable<Integer>,
Comparable<ListExt>
{ private Node first; private int size;
  public ListExt() {first = null;size = 0;}
  public int size(){ return size; }
  private Node node(int i) {assert i >= 0 && i < size;
   Node p = first;
   while (p != null && i > 0) {p = p.getNext();i--;}
   assert p != null;return p;}

   public int get(int i)          { return node(i).getElem(); }
   public void set(int i, int x) { node(i).setElem(x); }

    public void add(int i, int x) {assert 0 <= i && i <= size;
```

```
  size++;
  if (i == 0)
     first = new Node(x, first);
  else {Node prev = node(i - 1);
        prev.setNext(new Node(x, prev.getNext()));}}

 public int remove(int i) {assert 0 <= i && i < size;
 size--;
 if (i == 0) {int x = first.getElem();
  first = first.getNext();
  return x;}
  else {Node prev = node(i - 1);
        Node p = prev.getNext();
        prev.setNext(p.getNext());
        return p.getElem();}}

// Implementazione di Iterator<Integer>: ridenomiamo un
// elemento di ListExt come un elemento di ListIterator
// e implementiamo Iterator<Integer> in ListIterator
public ListIterator iterator(){return new ListIterator(first);}


// Implementazione di Comparable<ListExt>: confrontiamo due liste
// rispetto all'ordine lessicografico

public int compareTo(ListExt lista)
{Node p = this.first, q = lista.first;
//p, q = puntatori ai nodi delle due liste
//scorro le due liste un passo alla volta
 while ((p != null) && (q != null))
 {if (p.getElem() != q.getElem()) //le due liste sono diverse
   return p.getElem() - q.getElem(); //valore positivo se la prima
// lista e' piu' grande, negativo se e' piu' piccola
   else //vado avanti in entrambe le liste
    {p = p.getNext(); q = q.getNext();}}
// quando il while finisce ho esaurito almeno una delle due liste
// trovando solo elementi uguali. La lista finita prima e' minore
   if (p == null)   // la prima lista e' finita
    {if (q == null) // entrambe le liste sono finite
        return 0;   // quindi sono uguali
      else //prima lista finita ma seconda lista no
        return -1;} //prima lista minore
```

```
  else //prima lista NON finita, dunque seconda lista e' finita
    return +1;      // seconda lista minore
 }}}

//ListIterator.java Questa classe deve implementare Iterator<Integer>
import java.util.*;
// ListIterator e' un semplice indirizzo di un nodo
public class ListIterator implements Iterator<Integer>
{private Node next;
 public ListIterator(Node next){ this.next = next; }

/* per implementare Iterator<Integer> occorre fornire:
1. un metodo boolean hasNext() che ci dica se esiste un prossimo
oggetto della lista da visitare
2. un metodo Integer next() che sposti next sul prossimo oggetto da
visitare nella collezione e ne restituisca il valore, un intero. */
public boolean hasNext(){return  next != null;}

public Integer   next()
{int x = next.getElem(); //contenuto del prossimo nodo
 next = next.getNext();  //indirizzo del nodo dopo ancora
 return x;}} //x viene automaticamente trasformato da int a Integer

//TestList.java       Sperimentiamo foreach e compareTo
public class TestList {public static void main(String[] args) {
 ListExt a = new ListExt();
 for (int i = 20; i >= 0; i--) a.add(0, i);
 System.out.println( " Lista a={0,...,20}");
 for (Integer o : a) System.out.println(o);

 ListExt b = new ListExt();
 for (int i = 10; i >= 0; i--) b.add(0, i);
 System.out.println( " Lista b={0,...,10} ");
 for (Integer o : b) System.out.println(o);

 System.out.println( " a.compareTo(b) = " + a.compareTo(b));
 System.out.println( " b.compareTo(a) = " + b.compareTo(a));
 System.out.println( " b.set(7,100): ora b maggiore"); b.set(7,100);
 System.out.println( " Nuova Lista b: ora b_7=100");
 for (Integer o : b) System.out.println(o);
 System.out.println( " a.compareTo(b) = " + a.compareTo(b));
 System.out.println( " b.compareTo(a) = " + b.compareTo(a));}}
```

Lezione 20 Esempio di compito di esame

Lezione 20. Esercizio 1. Supponete sia già data la classe Node<T> per rappresentare nodi di liste concatenate su T generico, dove **null** denota la lista vuota. In Node<T> avete i metodi ***T getElem()*** e ***Node<T> getNext()***, e in T avete il test di eguaglianza ***x.boolean equals(T y)*** per ogni oggetto x!=null *(invece per x=null usate x==y).* Realizzate un metodo obbligatoriamente ricorsivo

public static <T> boolean inComune(Node<T> p, Node<T> q)

che restituisce **true** se esiste un elemento x che occorre in entrambe le liste p e q nella stessa posizione. Ad esempio, nel caso sia T=Integer, inComune(p,q) deve restituire

- **true** se p è **[5,7]** e q è **[8,7,1]**
- **true** se p è **[4,3,2]** e q è **[1,2,2]**
- **true** se p è **[1,2,3]** e q è **[1]**
- **false** se p è **[1,2]** e q è **[2,1]**
- **false** se una lista è vuota oppure lo sono entrambe.

Lezione 20. Esercizio 2. Date le classi

```
abstract class A {
public abstract void m1();
}

abstract class B extends A {
public void m1()       {System.out.println("B.m1");}
public abstract void m2(A obj);
}

class C extends B {
public void m1()       {System.out.println("C.m1");super.m1();}
public void m2(A obj) {System.out.println("B.m2");obj.m1();}
}
```

rispondere alle seguenti domande:

1. Se si eliminasse il metodo m2 dalla classe B, il codice che definisce A,B,C sarebbe comunque corretto? Perché?

2. Il seguente codice è corretto? Se no, spiegare perché. Se sì, determinare cosa stampa.

```
A obj2 = new B();
obj2.m2(obj2);
```

3. Il seguente codice è corretto? Se no, spiegare perché. Se sì, determinare cosa stampa.

```
A obj3 = new C();
obj3.m1();
```

4. Il seguente codice è corretto? Se no, spiegare perché. Se sì, determinare cosa stampa.

```
A obj4 = new C();
obj4.m2(obj4);
```

Lezione 20. Esercizio 3. Supponete sia già data la classe Node per rappresentare nodi di liste concatenate su interi, in cui **null** denota la lista vuota, e con gli usuali metodi get e set. Sia dato il codice:

```
public static void metodo(Node p)
{while (p != null)
 {if (p.getElem() < 0 && p.getNext().getElem() > 0)
    p.setNext(p.getNext().getNext());
  p = p.getNext();}}
```

1. Determinare sotto quali condizioni il metodo viene eseguito correttamente (cioè senza lanciare alcuna eccezione) e scrivere una corrispondente **asserzione** da aggiungere come precondizione per il metodo. Nello scrivere l'asserzione è possibile fare uso di eventuali metodi statici ausiliari che **vanno comunque definiti** anche se visti a lezione.

2. Descrivere in modo conciso e chiaro, in **non più di 2 righe di testo**, l'effetto del metodo.

Lezione 20. Esercizio 4. Date le classi

```
abstract class Tree {public abstract Tree insert(int elem);}

class Leaf extends Tree {
public Tree insert(int elem){ return new Branch(elem, this, this); }
public String toString(){ return "Leaf"; }}

class Branch extends Tree {
private int elem;private Tree left;private Tree right;
public Branch(int elem, Tree left, Tree right)
{this.elem = elem;this.left = left;this.right = right;}

public Tree insert(int elem){ System.out.println("CHECK POINT 2");
 if (elem < this.elem)      left = left.insert(elem);
 else if (elem > this.elem) right = right.insert(elem);
 return this;}

public String toString()
{return "Tree(" + this.elem + "," + left.toString() + "," +
          right.toString() + ")"; }}

public class Esercizio4 {public static void main(String[] args) {
 Tree t = new Leaf();
 t = t.insert(3);
 t = t.insert(1);
 t = t.insert(4);
 t = t.insert(2);
 System.out.println( "CHECK POINT 1");
 System.out.println(t.toString());}}
```

si disegni una rappresentazione dello stato della memoria (Stack e Heap)

1. al check point 1;
2. la prima volta che l'esecuzione raggiunge il check point 2.

Lezione 20: Soluzioni degli esercizi assegnati

Lezione 20. Soluzione Esercizio 1. Definiamo anche la classe Node<T>: in un esame dovete supporre che sia già data e non ridefinirla.

```
//Classe (non pubblica) nodi generici:
//null oppure un elemento di tipo T e l'indirizzo del prossimo nodo
 class Node<T> {private T elem; private Node<T> next;
 public Node(T elem, Node<T> next)
 {this.elem = elem;this.next = next;}
 public T getElem() {return this.elem;}
 public void setElem(T elem) {this.elem = elem;}
 public Node<T> getNext() {return this.next;}
 public void setNext(Node<T> next) {this.next = next;}

/* Metodo che controlla se due liste su T hanno almeno un elemento
   in comune nella stessa posizione. Prende x,y in una classe T e
   se x!=null usa x.equals(T y) per controllare se x,y sono uguali
   se x =null usa x==y (in questo caso equals non funziona) */
 public static <T> boolean inComune(Node<T> p, Node<T> q)
 {if ((p == null) || (q == null))
  return false; //non ci sono elementi in comune
  else {T x=p.getElem(), y=q.getElem();
        Node<T> l=p.getNext(), m=q.getNext();
  if (x==null)  //se x=null uso il test di eguaglianza ==
   return (x==y)||inComune(l,m); /* true se: il primo elemento e' in
comune oppure ci sono elementi in comune dopo */
  else          //se x!=null uso il test di eguaglianza equals
   return  (x.equals(y))||inComune(l,m);  }}  /*  true  se:  il  primo
elemento e' in comune oppure ci sono elementi in comune dopo */


//Main di prova: anche questo non e' richiesto all'esame.
 public static void main(String[] args)
 {Node<Integer> p = null, q = null, r = null;
  p = new Node<Integer>(1,new Node<Integer>(2,null));  // p = {1,2}
  q = new Node<Integer>(2,new Node<Integer>(2,null));  // q = {2,2}
  r = new Node<Integer>(2,new Node<Integer>(3,null));  // r = {2,3}
  System.out.println("p = {1,2}, q = {2,2}, r =  {2,3}\n");

  System.out.println("p={1,2} e q={2,2}:secondo elemento in comune");
  System.out.println(" inComune(p,q) = " + inComune(p,q)+ "\n");
  System.out.println("q={2,2} e r={2,3}:primo elemento in comune");
  System.out.println(" inComune(q,r) = " + inComune(q,r)+ "\n");
  System.out.println("r={2,3} e p={1,2}:nessun elemento in comune");
  System.out.println( " inComune(r,p) = " + inComune(p,r)+ "\n");}}
```

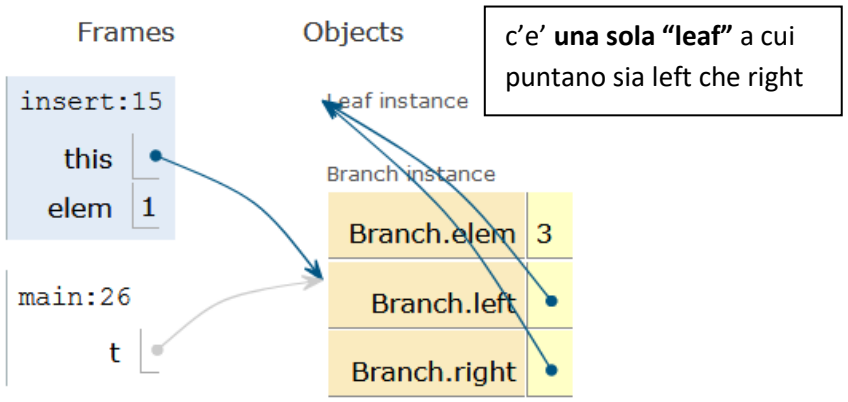
Lezione 20. Soluzione Esercizio 2. Ecco la risposta alle 4 domande. Forniamo anche una esecuzione di prova: a voi non è richiesta, dovete solo rispondere alle domande fatte.

```
abstract class A {
public abstract void m1();
}

abstract class B extends A {
public void m1()       {System.out.println("B.m1");}
public abstract void m2(A obj);
}

class C extends B {
public void m1()       {System.out.println("C.m1");super.m1();}
public void m2(A obj) {System.out.println("B.m2");obj.m1();}
}

class Test{public static void main(String[] args){
/* DOM1. Eliminare m2 in B non produce errori nelle classi A,B,C,
perche' in A,B,C non ci sono chiamate a m2 con tipo apparente B.
(Se invece nella definizione di A,B,C avessimo una chiamata b.m2(a);
con b di tipo B produrremmo un errore). */

/* DOM.2. Le prossime righe producono l'errore: "B is abstract;
cannot be instantiated" perche' cerchiamo di usare il costruttore
new B() della classe astratta B. */
/*  A obj2 = new B();
    obj2.m2(obj2); */

/* DOM.3 La classe C e' concreta e puo' produrre obj2 a cui si
assegna tipo apparente A. La classe A contiene m1. Durante
l'esecuzione, obj3 ha tipo esatto C, e m1 in C stampa "C.m1" e
richiama m1 in B che stampa "B.m1".Quindi le prossime righe stampano:
 C.m1
 B.m1 */
   A obj3 = new C();
   obj3.m1();

/* DOM.4 obj3 ha tipo esatto C e C contiene m2. Ma obj3 ha tipo
   apparente A che non contiene m2. Quindi inviare m2 a obj3 produce
   l'errore:
   "cannot find symbol.
    symbol: method m2(A) location: variable obj3 of type A"
/*  A obj4 = new C();
    obj4.m2(obj4); */
}}
```

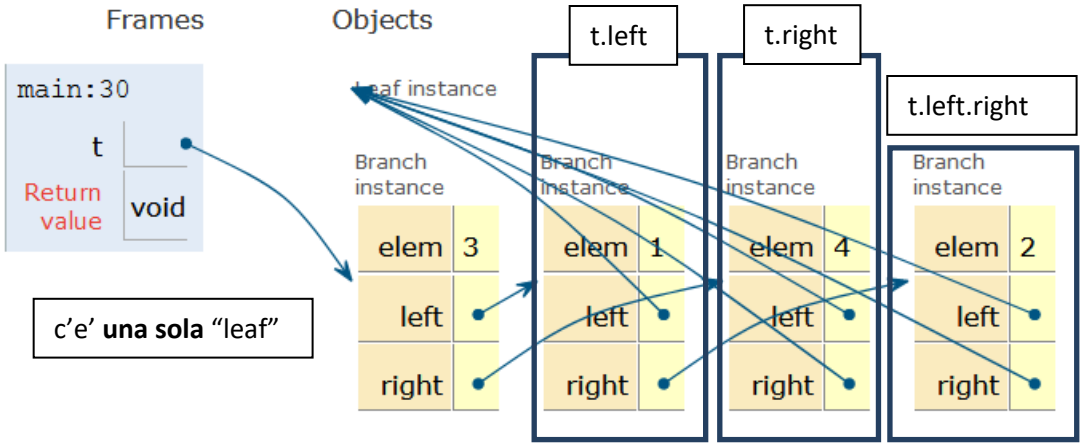
Lezione 20. Soluzione Esercizio 3. Ecco la descrizione del metodo e l'asserzione OK(p) che descrive quando il metodo solleva eccezioni su p. Definiamo anche la classe Node e un main di prova per l'asserzione richiesta. In un esame non viene richiesta né la classe Node né il main di prova.

```
class Node {private int elem;private Node next;
public Node(int elem, Node next) {this.elem = elem;this.next = next;}
public int  getElem(){return elem;}
public Node getNext(){return next;}
public void setElem(int  elem){this.elem = elem;}
public void setNext(Node next){this.next = next;}}

class Esercizio3 {public static boolean OK(Node p) //OK(p) controlla
// che ogni elemento negativo in p sia seguito da un altro elemento
  {while (p != null)
   {if (p.getElem() < 0 && p.getNext()==null)
//p ha un elemento negativo che non e' seguito da elementi
       return false;
     p=p.getNext();} //Se il while finisce, ogni elemento negativo in
// p e' seguito da un elemento
  return true;}

/* DESCRIZIONE metodo(p): per ogni elemento negativo in p, il metodo
elimina l'elemento seguente se positivo. metodo(p) solleva eccezione
se: un elemento negativo non ha seguente */
 public static void metodo(Node p) {assert OK(p):
 "C'e' un elemento negativo nella lista senza elemento seguente";
  while (p != null)
  {if (p.getElem() < 0 && p.getNext().getElem() > 0)
     p.setNext(p.getNext().getNext());
   p = p.getNext();}}

public static void scriviOutput(Node p)
{while(p!=null){System.out.println("  "+p.getElem());p=p.getNext();}}

public static void main(String[] args)
{Node p = new Node(0, new Node(-1,null)); //p={0,-1}
 Node q = new Node(0, new Node(-1,new Node(+1,null))); //q={0,-1,+1}
 System.out.println(" OK(p)=" + OK(p) + " OK(q)=" + OK(q));
 System.out.println(" p =");scriviOutput(p);
 if (!OK(p)) //p non viene accettata da OK(p) perche':
   System.out.println("l'elemento -1 non ha successore in p");
 System.out.println(" q ="); scriviOutput(q); metodo(q);
//metodo(q) cancella +1 perche' +1 segue -1 in q
 System.out.println( "dopo metodo(q): q="); scriviOutput(q);}}
```

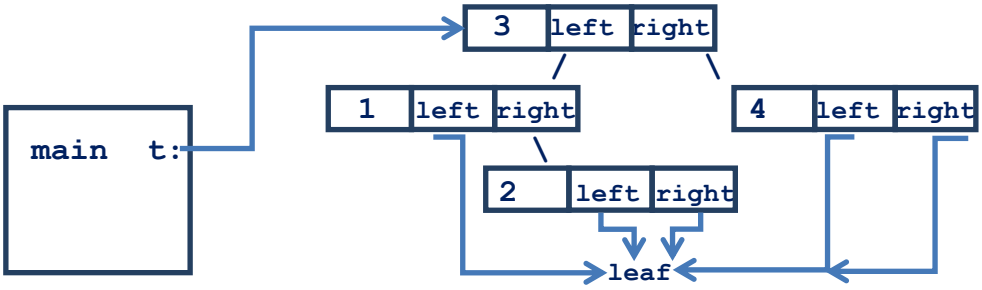
Lezione 20. Soluzione Esercizio 4. Il ***check point 2*** scatta la prima volta che dobbiamo aggiungere un nodo a un albero che ne contiene già uno, ma non l'abbiamo ancora fatto. Dunque abbiamo eseguito **`t = t.insert(3);`** e creato un albero con il solo nodo 3. Stiamo eseguendo **`t = t.insert(1);`** e stiamo per inserire 1 alla sinistra di 3. Ecco la situazione di Stack e Heap descritta da un Java visualizer.



Il ***check point 1*** scatta alla fine della costruzione di t, e subito prima della riga del main che stampa t. Ecco la situazione di Stack e Heap come viene descritta da un Java visualizer. Notiamo che c'è una sola ***"foglia"*** in memoria a cui puntano tutti i sottoalberi.



Ecco invece lo stesso disegno reso più leggibile, con t.left a sinistra e t.right a destra, e l'unica foglia in basso.



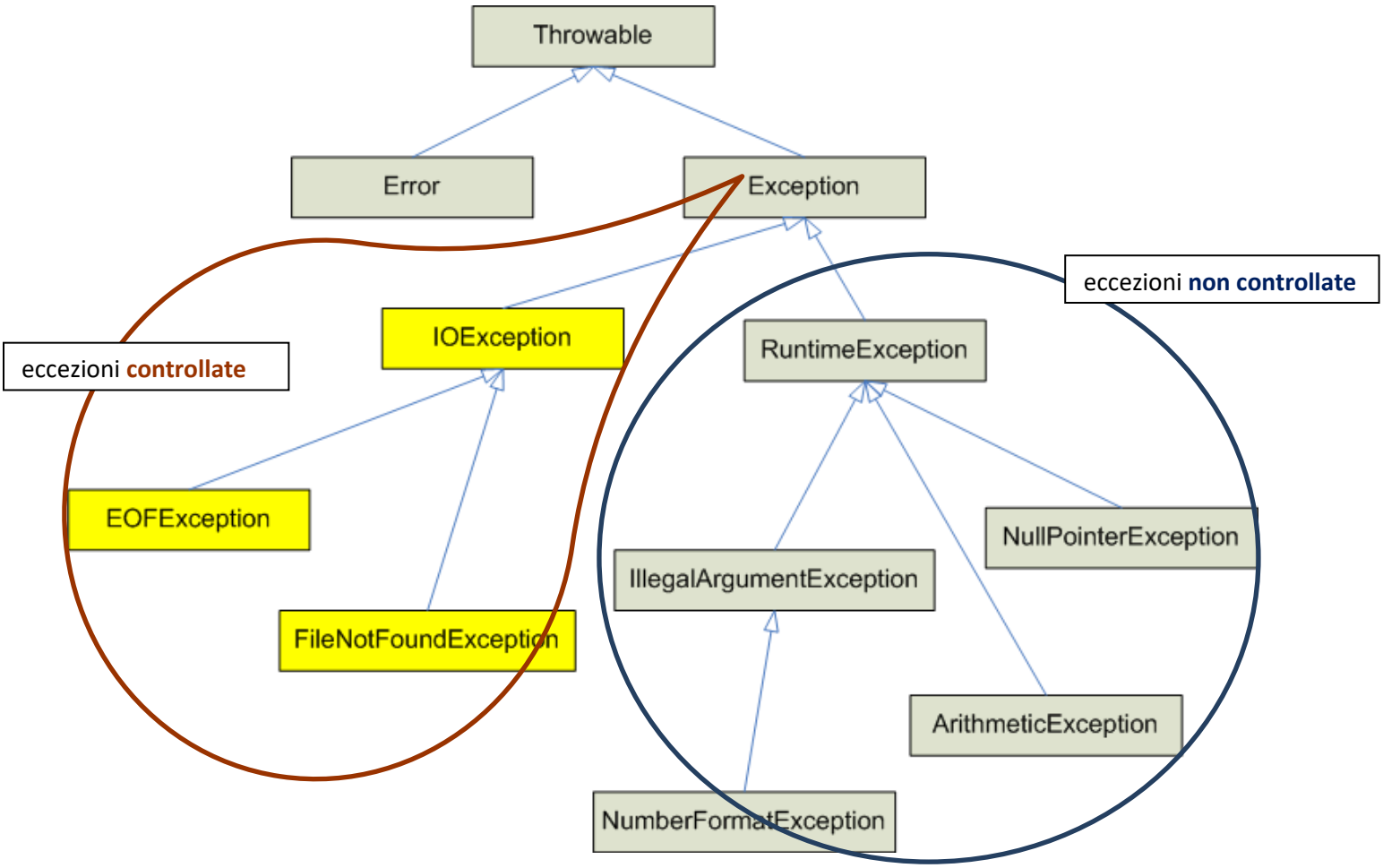
Lezione 21 Eccezioni controllate e non controllate

Eccezioni. Una eccezione in Java è un oggetto della classe Eccezione e viene usato per segnalare una situazione che il programma non sa gestire. Per esempio abbiamo le sottoclassi: ***ArithmeticException*** (divisione per zero, radice di un numero negativo), ***IllegalArgumentException*** (il metodo usato non è definito per quel valore di quel certo argomento), ***ArrayIndexOutOfBOundException*** (tentativo di accedere a un elemento **inesistente** di un vettore), ***NullPointerException*** (tentativo di accedere al contenuto dell'indirizzo null).

Vediamo qualche esempio di eccezioni.

```
//EsempiEccezioni.java Per diverse eccezioni comuni
//scriviamo un metodo che solleva quella eccezione
public class EsempiEccezioni {
 public static void test_ArithmeticException()
 {System.out.println(1 / 0);}

 public static void test_ClassCastException() {Object obj = "ciao";
Float f = (Float) obj;} //non e' possibile convertire String a Float

public static void test_NumberFormatException ()
{Integer.parseInt("ciao");} //parseInt trasforma una stringa in un
//intero, ma "ciao" non rappresenta un intero

 public static void test_IndexOutOfBoundsException()
 {int[] a = new int[10];a[10] = 0;} //a[10] non esiste

 public static void test_NullPointerException()
{Object obj = null;System.out.println(obj.toString());}
// Non possiamo mandare un metodo dinamico a null

    public static void main(String[] args) {
// Ognuna delle prossime righe solleva una eccezione
// test_ArithmeticException();
// test_ClassCastException();
// test_NumberFormatException();
// test_IndexOutOfBoundsException();
// test_NullPointerException();
}}
```

Eccezioni controllate e non controllate. Vediamo le relazioni di sottoclasse tra alcune classi di eccezioni.



Le classi di eccezioni incluse in ***RuntimeException*** sono, per esempio: IllegalArgumentException,NullPointerException, ArithmeticalException. Sono le eccezioni più comuni che si ***verificano al run-time***, e vengono dette ***eccezioni non controllate***. Tutte le eccezioni dell'esempio precedente sono in RuntimeException e non controllate. Le altre eccezioni sono dette ***eccezioni controllate***. Tra un attimo vediamo la differenza tra i due tipi di eccezioni.

Abbiamo già visto come sollevare delle eccezioni per terminare un programma, per esempio quelle sollevate da un **assert**. Sollevare una eccezione con un assert termina il programma con un messaggio di errore. Alcune sottoclassi di eccezioni hanno attributi, che si possono usare per scrivere il messaggio di errore.

Cattura di eccezioni. Non sempre è desiderabile terminare un programma in presenza di una eccezione. Per evitare di terminare un programma, le eccezioni possono essere ***catturate***: basta scrivere il codice nella forma ***try{…} catch(TipoEccezione e){…}***. Questo codice inizia eseguendo il corpo di ***try***, poi, se viene sollevata una eccezione di tipo TipoEccezioni, anziché terminare il programma esegue il corpo di ***catch***. Gli errori sono eccezioni non catturabili.

Possiamo scegliere quale tipo di eccezione sollevare scrivendo

throw new TipoEccezione(…);

Posso avvisare Java che un metodo ***void m()*** può sollevare una eccezione di tipo TipoEccezione aggiungendo alla segnatura del metodo, scrivendo: ***void m() throws TipoEccezione***. La cattura è possibile per una eccezione non controllata ed è necessaria per una eccezione controllata. In altre parole:

1. Quando ho una eccezione non controllata, non sono obbligato a scrivere throw, e non sono obbligato a catturare l'eccezione quando scrivo il metodo m().
2. Nel caso sollevi una eccezione controllata, sono obbligato a scrivere "throws TipoEccezione" nella segnatura di m(), e sono obbligato a catturare l'eccezione tutte le volte che uso m().

Possiamo definire noi stessi classi di eccezioni ***MiaEccezione*** estendendo una qualunque classe di eccezioni esistente: la situazione non cambia.

Posso catturare più eccezioni con le clausole ***try{…} catch(TipoEccezione_1 e_1){…}… catch(TipoEccezione_n e_n){…}.*** Ad essere eseguita è la prima clausola i tale che ***TipoEccezione_i*** sia il tipo dell'eccezione sollevata (oppure lo contenga).

Come usare le eccezioni. Le eccezioni ***non catturate*** rappresentano gli ***errori a cui non vogliamo, sappiamo o possiamo rimediare***. Quando sviluppiamo un programma utilizziamo gli assert per sollevare eccezioni non controllate, terminare il programma e tracciare l'errore. Le ***eccezioni catturate*** rappresentano le ***procedure di emergenza*** che uso quando incontro un'eccezione, voglio avvisare dell'errore ma ***evitare di spegnere il programma***. Per esempio posso interrompere il solo calcolo che sto facendo, segnalare il tipo di errore insieme con i suoi parametri, e continuare con il resto del calcolo. Per esempio: questa scelta è necessaria quando sto scrivendo

un programma che *non può essere spento immediatamente in condizioni di sicurezza* (guida di un'automobile, di un aereo o di un drone), oppure che *spento completamente produrrebbe danni economici* (un sistema di prenotazioni o di pagamenti elettronici), o quando *l'errore dipende dai dati inseriti* (chiediamo a una applicazione di aprire un file che non c'è). Non abbiamo il tempo di presentare esempi rilevanti di eccezioni nel corso e ci limitiamo a spiegare come si scrivono le eccezioni in Java.

Esempi di eccezioni non controllate *(cattura possibile)*. Vediamo ora come sostituire l'uso di assert con il lancio di eccezioni non controllate. Riprendiamo alcune classi viste nelle lezioni precedenti. Rivediamo la classe Bottiglia (Lezione 05), e rimpiazziamo l'assert con il lancio di una eccezione di tipo ***IllegalArgumentException***, dunque non controllata, quando incontriamo un errore. Questa eccezione può essere catturata oppure no.

```
public class Bottiglia {private int capacita; // 0 <= capacita
 private int livello;  // 0 <= livello <= capacita

// IllegalArgumentException e' una eccezione non controllata,
// dunque se la sollevo NON sono obbligato ad aggiungere
// "throws IllegalArgumentException" alla segnatura di
// Bottiglia(int capacita), e non sono obbligato a catturarla

  public Bottiglia(int capacita) {
 if (capacita < 0)
  throw new IllegalArgumentException("capacita negativa: "+capacita);
 this.capacita = capacita; this.livello = 0;}

  public int getCapacita() {return this.capacita;}
  public int getLivello() {return this.livello;}

  public void aggiungi(int quantita) {if (quantita < 0)
   throw new IllegalArgumentException("aggiungi: quantita negativa");
  livello = Math.min(livello + quantita, capacita);}

  public int rimuovi(int quantita) {if (quantita < 0)
   throw new IllegalArgumentException("rimuovi: quantita negativa");
 int rimossa = Math.min(quantita, livello);
 this.livello -= rimossa;
 return rimossa;}}
```

Vediamo ora come catturare (oppure non catturare) le eccezioni sollevate nella classe Bottiglia.

```
public class TestBottiglia {
public static void main(String[] args) {// una capacita' negativa
// causa il lancio dell'eccezione non controllata
// IllegalArgumentException nel costruttore di Bottiglia
  try{new Bottiglia(-10);}
  catch(IllegalArgumentException e) //Catturo l'eccezione e la scrivo
      {System.out.println(e.toString());}

// dato che IllegalArgumentException non e' controllata NON
// sono obbligato a catturare l'eccezione: se non lo faccio
// una eccezione ferma il programma.
 System.out.println
  ("\nLa prossima richiesta fa cadere il programma\n");
 new Bottiglia(-5);}}
```

Un esempio simile al precedente è la classe Stack degli stack di interi (Lezione 04). Anche in questo caso sostituiamo l'assert con il lancio una eccezione non controllata, non siamo obbligati a indicare che la lanciamo usando un "throws", e non siamo obbligati a catturarla.

```
public class Stack {private int[] stack; // stack != null
                    private int dim;    // 0 <= dim <= stack.length

// IllegalArgumentException e' una eccezione non controllata,
// dunque se la sollevo NON sono obbligato ad aggiungere
// "throws IllegalArgumentException" alla segnatura di
// Stack(int dim), e non sono obbligato a catturare
// l'eccezione ogni volta che uso Stack(int dim)

public Stack(int capacita) {
 if (capacita < 0)
   throw new IllegalArgumentException("capacita' stack negativa");
 this.stack = new int[dim];this.dim = 0;}

public boolean vuota() {return this.dim == 0;}
public boolean piena() {return this.dim == this.stack.length;}
```

```
public void push(int x) {if (piena())
      throw new IllegalStateException("stack pieno");
    stack[dim++] = x;}
public int pop() {if (vuota())
   throw new IllegalStateException("stack vuoto");
 return stack[--dim];}}
```

Proviamo a definire noi stessi una classe di eccezioni. Se scegliamo una classe di eccezioni non controllate, quando la lanciamo possiamo scegliere se indicarla nella segnatura del metodo in cui la lanciamo con un "throw", e non siamo obbligati a catturarla.

```
public class MaxSuAlberoVuoto extends RuntimeException {
 public MaxSuAlberoVuoto(String msg) {super(msg);}}

// Implementazione della classe Leaf per rappresentare alberi vuoti

public class Leaf extends Tree {public Leaf() { }
 public boolean empty() {return true;}

// MaxSuAlberoVuoto e' non controllata, dunque se la sollevo
// posso scegliere se aggiungere "throws MaxSuAlberoVuoto" alla
//segnatura di max(). Non sono obbligato a catturare
//l'eccezione ogni volta che uso max()

 public int max() throws MaxSuAlberoVuoto {
   throw new MaxSuAlberoVuoto("max su Leaf");} //Notiamo che
 // non e` piu` necessario inserire al fondo un "return 0;"
 // in quanto il compilatore Java e' a conoscenza del fatto
 // che throw causa la terminazione del metodo corrente

public boolean contains(int x) {return false;}

public Tree insert(int x) {return new Branch(x, this, this);}}
```

Esempi di eccezioni controllate *(cattura necessaria)*. Proviamo a definire un'eccezione controllata, per impedire la creazione di una frazione di denumeratore ≤ 0, ma senza far cadere il programma quando questo capita. Per le eccezioni controllate siamo obbligati a scrivere "throws", e poi a catturarle. Definisco una classe di eccezioni controllate a partire dalla classe Exception. Aggiungo come informazione il denominatore della frazione rifiutata.

```
import java.util.*;

class DenZeroException extends Exception //controllata
{private int den; //den viene rifiutato quanto den<=0
 public DenZeroException(int den){this.den=den;}
 public int getDen(){return den;}}

class Frazione {private int num; private int den;
// Invariante classe: il denominatore deve essere > 0
public Frazione(){num=1; den=1;} //costruttore pubblico:
//restituisce un valore di default = 1/1

// uso un costruttore privato, puo' creare frazioni
// senza senso, ma non e' accessibile dall'esterno
private Frazione(int n, int d) {num = n;den = d;}

// Inserisco l'eccezione in un metodo create pubblico
// che non chiamo costruttore. create usa il costruttore privato
// e se necessario solleva un'eccezione
public static Frazione create(int n, int d) throws DenZeroException
{if (d<=0) throw new DenZeroException(d);
 return new Frazione(n,d);}

// DenZeroException e' una eccezione controllata, se la sollevo
//sono obbligato ad aggiungere "throws DenZeroException" alla
//segnatura di create, e sono obbligato a catturare l'eccezione
//ogni volta che uso create

 public String toString(){return num + "/" + den;}}

//Quando uso il metodo create devo inserire il metodo dentro un "try"
//e aggiungere alla fine un "catch" per trattare il caso
//dell'eccezione di tipo "DenZeroException"

public class ProvaFrazione
{public static void main(String[] args){
 Scanner scanner = new Scanner(System.in);//automa per leggere input
 boolean done    = false;
 int n, d; Frazione f = new Frazione(); //f=default=1/1
//f = Frazione.create(2,3); //NO: solleva come eccezione: "unreported
//exception",non ho catturato DenZeroException
```

```
 while (!done)
 try {System.out.println("Inserisci il numeratore:");
    n = scanner.nextInt();
    System.out.println("Inserisci il denominatore:");
    d = scanner.nextInt();
    f = Frazione.create(n,d); //posso usare create solo dentro try{}
    done = true;}
catch (DenZeroException err) //Se leggo un d<=0 chiedo di nuovo
   {System.out.println
    ("Den. " + err.getDen() + "<= 0!. Inserisci ancora:");}
System.out.println("Hai inserito " + f);}}
```

"**Catch" ripetuti.** Vediamo infine un esempio di cattura di diverse eccezioni (non controllate), con l'uso di diversi catch.

```
public class TestTryCatch {
 public static void m() //solleva una IllegalStateException
  {throw new IllegalStateException( "non dovevi chiamarmi");}

 public static void main(String[] args) {
  try{
     m();//otteniamo una IllegalStateException: prossima riga saltata
     System.out.println( "ce l'ho fatta! ho la risposta di m()");}

  catch (RuntimeException e) { //Questa clausola scatta perche'
// RuntimeException include IllegalStateException
     System.out.println( "catturata IllegalStateException");}

  catch (Exception e) { //Questo catch verrebbe raggiunto da
// eccezioni diverse da IllegalStateException
     System.out.println( "catturata eccezione " + e);}

  finally { //qui volendo posso aggiungere una clausola
 //che sara' eseguita sia che arriviamo da try che da un catch
     System.out.println( "clausola finale");}}
}
```

Lezione 22 Esempi di uso di Iterable e di eccezioni controllate

Lezione 22. Parte 1. La classe IOException. Nella prima parte della lezione consideriamo un esempio di eccezioni controllate di maggiori dimensioni, che riguarda una IOException. In particolare consideriamo il caso in cui un input non arriva o arriva errato.

Consideriamo (in versione giocattolo) un passo preliminare della compilazione, il problema di raggruppare in "token" i caratteri che compongono una istruzione di un linguaggio di programmazione. Un token è la coppia di una "Tag", una etichetta che dice a cosa serve una parte di una istruzione, e della sottostringa che compone quella parte di istruzione. Consideriamo le seguenti tag:

```
//Tag.java

public enum Tag {
//etichette che indicano l'uso di parti di una istruzione
 NUM,                        // numero
 ID,                         // identificatore (costante o variabile)
 LPAR, RPAR,                 // parentesi aperta e chiusa
 PLUS, MINUS,TIMES,DIVIDE,   // diverse operazioni ...
 ASSIGN,SEMICOLON, EOF}      // End Of File: un token per terminare
```

Isolare un testo in "token" e' il primo passo per capirlo. Nel file Tag.java abbiamo inserito una "tag" per i numeri, i nomi delle operazioni e cosi' via. Per esempio: nell'istruzione

perimetro=(bMin+bMag)*h/2;

possiamo isolare i seguenti "token":

<ID,"perimetro">, <LPAR,"(">, <ID,"bMin">, <PLUS,"+">, …

Definiamo una classe Token dei token. Quindi, per fare una semplice prova, costruiamo dei token e ne stampiamo la descrizione. Normalmente, invece, i token non vengono costruiti ma isolati dentro a un testo.

```
//Token.java
public class Token { private Tag tag; private String text;
 public Token(){}
 public Token (Tag tag, String text)
```

```
  {this.tag = tag;this.text = text;}
 public Tag    getTag()  {return tag;}
 public String getText() {return text;}
//niente metodi set: non ho bisogno di modificare un token

//Produco una descrizione di un Token come stringa
 public String toString()
 {return "--------------- >  <" + this.tag + ", " + this.text + ">";}

 public static void main(String[] args) {
  Token prova1 = new Token(Tag.NUM, "123");
  Token prova2 = new Token(Tag.ID, "pippo");
  System.out.println(prova1);
  System.out.println(prova2);
 }
}
```

Definisco una classe "Lexer" di operazioni che isolano dei token all'interno di una istruzione di un linguaggio di programmazione. L'esempio e' semplicissimo: consideriamo solo le quattro operazioni su interi naturali, senza parentesi, ma con EOF. I token saranno gli interi naturali, i nomi delle 4 operazioni, ed EOF. Ci sono diverse eccezioni da gestire: la fine del testo, che deve diventare un token special "EOF", e la presenza di simboli non previsti (in questo caso lanciamo una eccezione controllata ***IOException***, che dovremo poi catturare).

Attributi e metodi di Lexer sono:

1. **char peek.** È l'unico attributo di un lettore di token, e rappresenta il carattere all'inizio del prossimo token da leggere. All'inizio e ogni volta che finisco il testo pongo peek=' '.
2. ***void readch().*** Legge un carattere da tastiera se c'è, altrimenti ferma l'esecuzione finche' noi non inseriamo nuovi caratteri da tastiera. readch() usa la primitiva System.in.read(), e quest'ultima se la lettura del carattere fallisce lancia una IOException, controllata. readch() cattura questa eccezione e la rimpiazza con il carattere EOF=-1.
3. **Token scan() throws IOException.** Raggruppa in un token i caratteri da peek in poi. Poi scan() porta peek al primo carattere dopo il token. Se il carattere trovato non è tra quelli previsti, scan() lancia una IOException, controllata, che deve venir trattata da chi usa scan().

```
//Lex.java
import java.io.*; // Per avere la classe IOException, che altrimenti
// devo descrivere come: java.io.IOException

public class Lexer { char peek = ' '; //valore iniziale peek:
//riassegno peek=' ' quando finiscono i token

public void readch()
 {try {peek = (char) System.in.read();} //puo' lanciare IOException
  catch (IOException exc)
  {peek = (char) -1; }}//Rappresento l'eccezione con il carattere -1.

 public Token scan() throws IOException{
//Salto il valore iniziale di peek e gli spazi bianchi nell'input
  while (peek == ' ' || peek == '\t' || peek == '\n'){readch();}
//Leggo il primo pezzo significativo: il carattere -1 (End Of File)
//una operazione +,*,-,/, oppure un intero. Se raggiungo EOF pongo
// peek=' '. Se trovo una operazione mando avanti peek di 1.
 switch (peek) {
  case (char) -1: peek=' '; return new Token(Tag.EOF,    "");
  case '+':       readch(); return new Token(Tag.PLUS,   "+");
  case '*':       readch(); return new Token(Tag.TIMES,  "*");
  case '-':       readch(); return new Token(Tag.MINUS,  "-");
  case '/':       readch(); return new Token(Tag.DIVIDE, "/");
  default:
/* Se trovo una cifra, raccolgo tutte le cifre consecutive che trovo
per formare una stringa che etichetto come NUM (intero). Mi fermo al
primo carattere dopo il token e NON mando piu' avanti peek. */
   if (Character.isDigit(peek))
  {String s = "";
   do {s = s + peek; readch();} while (Character.isDigit(peek));
//Restituisco la stringa raccolta, e la etichetto come NUM
    return new Token(Tag.NUM, s);}
   else {peek=' '; //fine token, riassegno peek al valore iniziale
throw new IOException( "Carattere inserito ne' cifra ne' +*-/");}}}

//NOTA. Un metodo che usa un metodo che solleva eccezioni
//deve a sua volta essere commentato con il "throws"
//altrimenti non viene accettato. Per esempio devo scrivere:

 public Token scan2() throws IOException {return scan();}
```

```
//Come esperimento, inserisco una stringa e la scompongo in token
//Se la stringa contiene un carattere diverso da una cifra oppure una
//delle 4 operazioni +.*,-,/ allora scan() solleva una eccezione che
//inserendo un messaggio e uscendo dal while

 public static void main(String[] args) {
  Lexer lex = new Lexer();
  Token t = new Token();
  System.out.println( "Scrivere espressione con: naturali +-*/");
  System.out.println( "Per finire inserite qualunque altra cosa");

  while (t.getTag() != Tag.EOF) /* Uso scan() finche' fallisce la
ricerca di un token. Non stampo il fallimento. */
   try {t = lex.scan(); System.out.println( "Token: " + t);}
   catch(IOException e)
// quando viene inserito un carattere non previsto termino il ciclo
   {System.out.println(e.getMessage()); t = new Token(Tag.EOF,"");}}}
```

Lezione 22. Parte 2. Un nuovo esempio di uso dell'interfaccia Iterable<T>.

Definiamo una classe ***Interval*** di "intervalli" [first, last[tra due nodi first e last che implementa Iterable<Integer>. [first,last[e' la lista di primo nodo first e senza i nodi della lista dal nodo last in poi (last escluso). **Esempi.** Se first={1,2,3} allora [first,null[={1,2,3},[first,first[={} mentre se last=terzo nodo di first, allora [first,last[={1,2} (scarto il terzo nodo, contenente il 3). **Proprietà.** Interval consente esclusivamente l'uso dell'istruzione *"foreach"*: ***for(Integer n:interval){ … n … },*** dove interval = [p,q[. Il vantaggio di un *"foreach"* è consentire di scrivere un ciclo for su [p,q[senza rendere pubblici gli indirizzi p,q, e senza doversi preoccupare del dettaglio che il ciclo for deve fermarsi se arriva a q (ci pensa il foreach). La classe Interval ***non consente di accedere*** ai nodi della lista p posti dal nodo q in poi.

```
import java.util.*; //per le interfaccie
class Node {private int elem;private Node next;
public Node(int elem, Node next) {this.elem = elem;this.next = next;}
public int  getElem()             {return elem;}
public Node getNext()             {return next;}
public void setElem(int  elem)    {this.elem = elem;}
public void setNext(Node next)    {this.next = next;}}
```

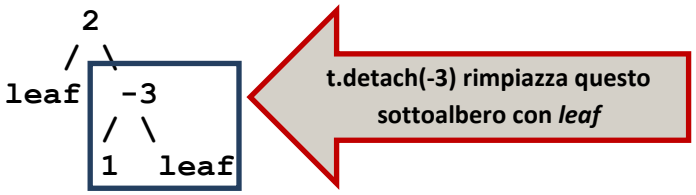
```
class Interval implements Iterable<Integer>{private Node first, last;
  public Interval(Node first, Node last)
   {this.first=first;this.last=last;}
  public IntervalIterator iterator()
   {return new IntervalIterator(first,last);}}

class IntervalIterator implements Iterator<Integer>
{private Node first,last;
 public IntervalIterator(Node first,Node last)
  {this.first=first;this.last=last;}

// [first,last[ ha almeno un nodo se first!=last e first!=null
 public boolean hasNext(){return first!=last && first!=null;}

 public Integer next() //portiamo avanti first senza cambiare last
//e restituiamo il contenuto del first originario
 {int elem=first.getElem();first=first.getNext();return elem;}}

//Adesso posso usare l'istruzione: for(Integer n:interval){...n...}
public class TestIterator{public static void main(String[] args){
  System.out.println( "p={1,2,3,4}");
  Node p=new Node(1,new Node(2, new Node(3, new Node(4,null))));
  System.out.println( "q=quarto nodo di p");
  Node q=p.getNext().getNext().getNext(); //salto i primi 3 nodi di p

  System.out.println( "\n [p,q[=i primi tre nodi di p");
  Interval interval=new Interval(p,q);
  System.out.println( "stampo [p,q[={1,2,3}");
  for(Integer n:interval)System.out.println(n);

  System.out.println( "\n [p,null[=p");
  Interval interval2=new Interval(p,null);
  System.out.println( "stampo [p,null[=p={1,2,3,4}");
  for(Integer n:interval2) System.out.println(n);

  System.out.println( "\n [p,p[={}");
  Interval interval3=new Interval(p,p);
  System.out.println( "stampo [p,p[={}");
  for(Integer n:interval3) System.out.println(n);}}
```

Lezione 23 Esempio di compito di esame

Lezione 23. Esercizio 1. Siano date le classi (incomplete):

```
abstract class Tree<T> {public abstract Tree<T> detach(T x);}

class Leaf<T> extends Tree<T> {
public Tree<T> detach(T x) { /* COMPLETARE */ }}

class Branch<T> extends Tree<T> {private T elem;private Tree<T>
left;private Tree<T> right;
public Branch(T elem, Tree<T> left, Tree<T> right)
{this.elem = elem;this.left = left;this.right = right;}
public Tree<T> detach(T x) { /* COMPLETARE */ }}
```

Fornire le implementazioni del metodo detach in Leaf e Branch in modo tale che t.detach(x) restituisca una versione modificata di t in cui ogni sottoalbero avente radice x è stato sostituito con l'albero vuoto ***leaf***. Se x non è presente, il metodo deve restituire l'albero invariato. **Esempio.** Sia t l'albero



Sostituzione di un elemento di un albero tramite il metodo detach

t.detach(-3) deve restituire l'albero che contiene il solo elemento 2. Realizzare il metodo detach in modo da minimizzare il numero di nuovi oggetti creati. Non è consentito aggiungere metodi, né usare cast o metodi della libreria standard di Java. Ricordatevi che per confrontare x con y in T dovete usare ***"equals"*** se x≠null, e ***"=="*** se x=null.

Lezione 23. Esercizio 2.

```
interface I {public void m1(J obj);}

interface J {public void m2();}

abstract class C implements I {public abstract void m1(J obj);}

class D extends C implements J {
 public void m1(J obj)
  {if (this != obj) obj.m2(); System.out.println("D.m1");}
 public void m2()
  {System.out.println("D.m2");m1(this);}}
```

Rispondere alle seguenti domande:

1. Se si eliminasse il metodo m1 dalla classe C, il codice sarebbe comunque corretto? Perché?

2. Il seguente codice è corretto? Se no, spiegare perché. Se sì, determinare cosa stampa.

```
I obj2 = new D();
((D) obj2).m2();
```

3. Il seguente codice è corretto? Se no, spiegare perché. Se sì, determinare cosa stampa.

```
J obj3 = new D();
C x = (C) obj3;
x.m1(new D());
```

4. Il seguente codice è corretto? Se no, spiegare perché. Se sì, determinare cosa stampa.

```
C obj4 = new D();
obj4.m1(obj4);
```

Lezione 23. Esercizio 3.

```
public class Node<T> {public T elem; public Node<T> next;
public Node(T elem, Node<T> next)
 {this.elem = elem;this.next = next;}
public static <T extends Comparable<T>> void metodo(Node<T> p, T x)
 {while (x.compareTo(p.elem) < 0) p = p.next;
  p.next = null;}}
```

1. Determinare sotto quali condizioni il metodo viene eseguito correttamente (cioè senza lanciare alcuna eccezione) e scrivere una corrispondente **asserzione** da aggiungere come precondizione per il metodo. Nello scrivere l'asserzione è possibile fare uso di eventuali metodi statici ausiliari che **vanno comunque definiti** anche se visti a lezione.
2. Descrivere in modo conciso e chiaro, in **non più di 2 righe di testo**, l'effetto del metodo.

Lezione 23. Esercizio 4. Si disegnino Stack e Heap al raggiungimento del checkpoint 1 e del checkpoint 2 per il codice qui sotto.

```
abstract class List {public abstract List reverse(List prev);}

class Nil extends List {public List reverse(List prev)
{System.out.println( "CHECK POINT 2"); return prev;}}

class Cons extends List {private int elem;private List next;
public Cons(int elem, List next) {this.elem = elem;this.next = next;}
public List reverse(List prev)
{List next = this.next;this.next = prev;return next.reverse(this);}}

public class TestHeap {public static void main(String[] args) {
List l = new Cons(1, new Cons(2, new Nil()));
System.out.println( "CHECK POINT 1");
l = l.reverse(new Nil());}}
```

Nota. L'esecuzione di ***this.reverse(prev)*** con reverse definito qui sopra rovescia tutte le freccie nella lista this, e manda la freccia che partiva dal primo elemento di *this* a puntare verso prev. **Esempio.** Se this={1,2}, e nil è l'unico elemento di Nil, allora eseguendo this.reverse(prev) otteniamo:

Rappresentazione grafica del this reverse

Lezione 23. Soluzione esercizio 1.

```
abstract class Tree<T> {public abstract Tree<T> detach(T x);}
class Leaf<T> extends Tree<T> {public Tree<T> detach(T x) {return
this;}}
class Branch<T> extends Tree<T> {private T elem;
private Tree<T> left; private Tree<T> right;
public Branch(T elem, Tree<T> left, Tree<T> right) {this.elem = elem;
this.left = left;this.right = right;}

public Tree<T> detach(T x)
 {if (((x==null)&&(x==elem)) || ((x!=null)&&x.equals(elem)))
    return new Leaf<T>();
  else {left=left.detach(x);right=right.detach(x);return this;}}}
```

Lezione 23. Soluzione Esercizio 2.

Domanda 1. Se si eliminasse il metodo m1 dalla classe C, il codice sarebbe comunque corretto?
Risposta 1. Sì, perché la classe C è astratta e dunque non é tenuta a fornire une implementazione per tutti i metodi elencati nelle interfaccie che implementa. Inoltre, C eredita m1 che dunque è presente per gli oggetti di tipo apparente incluso in C, ed m1 è astratto, dunque le sottoclassi di C (in particolare, D) non lo possono invocare con un "super".

Per le altre domande forniamo un main di prova.

```
interface I {public void m1(J obj);}
interface J {public void m2();}
abstract class C implements I {public abstract void m1(J obj);}
class D extends C implements J {
 public void m1(J obj)
  {if (this != obj) obj.m2(); System.out.println("D.m1");}
 public void m2()
  {System.out.println("D.m2");m1(this);}}

public class Esercizio2{public static void main(String[] args){
System.out.println( " Domanda 2");
I obj2 = new D();
((D) obj2).m2();   /*Risultato: D.m2 D.m1 */

System.out.println( " Domanda 3");
J obj3 = new D();
C x = (C) obj3;
x.m1(new D());   /*Risultato: D.m2 D.m1 D.m1 */

/* System.out.println( " Domanda 4");
   C obj4 = new D();
   obj4.m1(obj4);                    */
/* Risposta. Si ottiene l'errore: no suitable method found for m1(C)
    method I.m1(J) is not applicable
      (argument mismatch; C cannot be converted to J)
    method C.m1(J) is not applicable
      (argument mismatch; C cannot be converted to J) */
}}
```

Lezione 23. Soluzione Esercizio 3.

Risposta 1. Il metodo lancia un'eccezione se tolti tutti gli elementi > x si ottiene la lista vuota (in particolare, se x=null). Qui sotto esprimiamo descriviamo con un metodo statico:

<T extends Comparable<T>> boolean OK(Node<T> p, T x)

il caso in cui non ci sono eccezioni.

Risposta 2. Il metodo tronca la lista p dopo la prima occorrenza di un elemento che e' ≤ x.

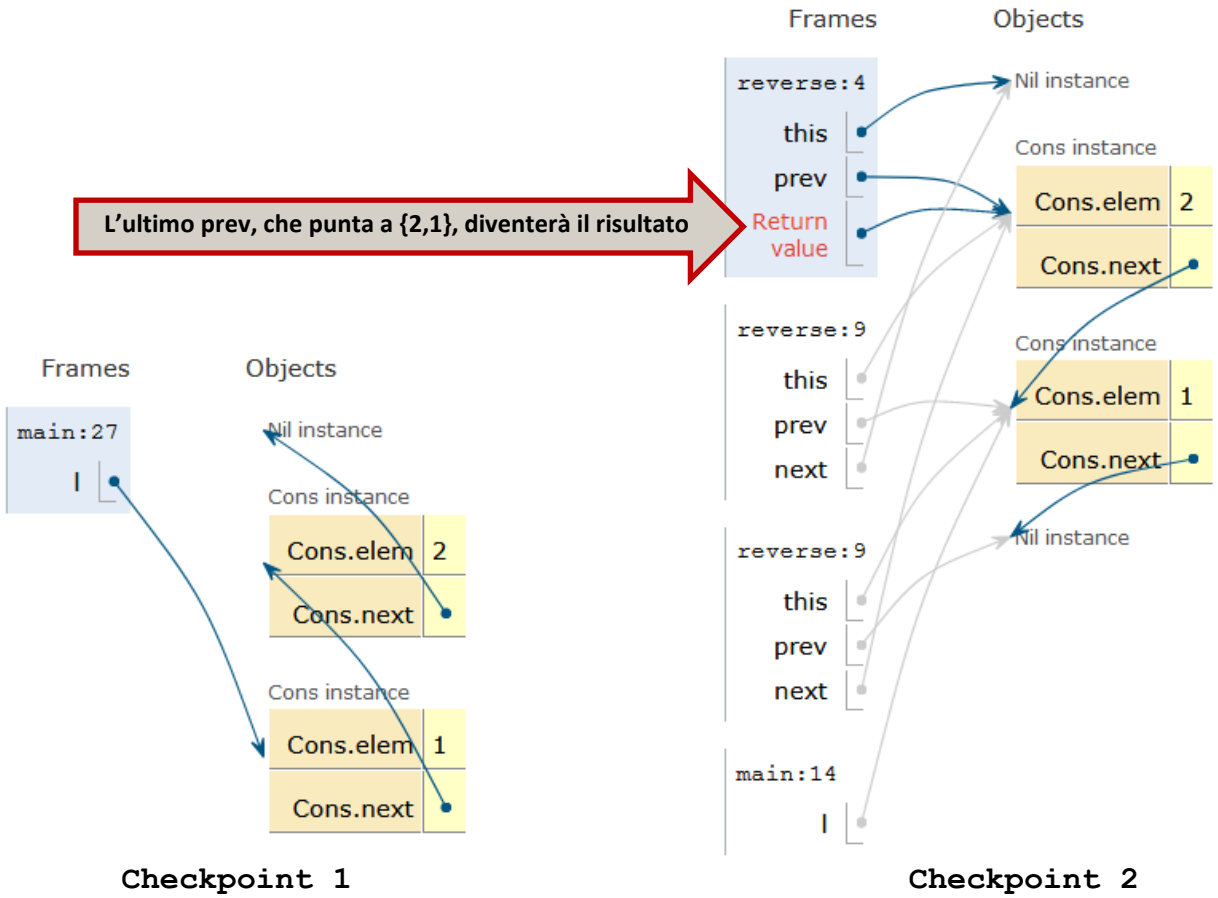
```
public class Node<T> {public T elem; public Node<T> next;
public Node(T elem, Node<T> next){this.elem = elem;this.next = next;}
public static void scriviOutput(Node p)
{while(p!=null){System.out.println(p.elem);p=p.next;}}

public static <T extends Comparable<T>> void metodo(Node<T> p, T x)
 {assert OK(p,x); //Per prevenire una NullPointerException
  while (x.compareTo(p.elem) < 0) p = p.next;
  p.next = null;}

public static <T extends Comparable<T>> boolean OK(Node<T> p, T x)
 {if (x==null) return false;
  while ((p!=null) && (x.compareTo(p.elem) < 0) )
    p = p.next;
  return p!=null;}

//Main di prova
public static void main(String[] args){Node<Integer>
   p= new Node<>(30, new Node<>(20, new Node<>(10, null)));
System.out.println("p=");scriviOutput(p);
System.out.println("\n OK(null,30)="+OK(null,30));//OK(null,30)=false
System.out.println("\n OK(p,null)="+OK(p,null));  //OK(p,null)=false
System.out.println("\n OK(p,10)=" + OK(p,10));    //OK(p,10)=true
System.out.println("applico metodo(p,10): p non cambia");
metodo(p,10); scriviOutput(p);
System.out.println("\n OK(p,20)=" + OK(p,20));    //OK(p,20)=true
System.out.println("applico metodo(p,20): ora p={30,20}");
metodo(p,20); scriviOutput(p);
System.out.println("\n adesso OK(p,10)="+OK(p,10));//OK(p,10)=false
System.out.println("se applico metodo(p,10) sollevo un'eccezione");
  /* System.out.println("applico metodo(p)");metodo(p,10);
     System.out.println("p=");scriviOutput(p); */   }}
```

Lezione 23. Soluzione esercizio 4.

Nell'esempio dato abbiamo ***prev=nil=new Nil()***, dunque l'esecuzione di *this*.reverse(prev) rovescia tutte le freccie nella lista *this*, e manda la freccia che partiva dal primo elemento di this a puntare verso un nuovo *nil*.

1. Quando l'esecuzione inizia ***(checkpoint 1)*** abbiamo l={1,2} nella heap.

2. Quando arriviamo al ***checkpoint 2*** troviamo nello stack due chiamate a Cons.reverse, con *this*=nodo 1, *this*=nodo 2, e prev, next uguali, rispettivamente, al nodo immediatamente prima di *this* e immediatamente dopo *this* nel risultato del metodo. In cima allo stack troviamo una chiamata a Nil.reverse. Compito di questa chiamata è dichiarare che l'ultimo valore di prev, che punta al nodo 2, è il risultato: prev={2,1}.



Rappresentazione grafica dell'algoritmo di risoluzione dell'esercizio con i suoi checkpoint

Lezione 24 Visita in pre-, in-, post-ordine e per livelli

In questa lezione presentiamo dei metodi che visitano un albero secondo diversi ordini stampando i nodi: pre-ordine, in-ordine, post-ordine e per livelli. Le prime tre visite sono definite ricorsivamente.

1. Nel ***pre-ordine*** visitiamo prima la radice, poi il sotto-albero sinistro e infine il sotto-albero destro.
2. Nell'***in-ordine*** visitiamo prima il sotto-albero sinistro, poi la radice e infine il sotto-albero destro.
3. Nel ***post-ordine*** visitiamo prima il sotto-albero sinistro, poi il sotto-albero destro e infine la radice.

Queste visite sono anche dette visite "in depth" o in profondità, perché esplorando completamente un ramo prima di passare al ramo accando. La ***visita per livelli*** viene definita tramite un ciclo: visitiamo prima la radice, poi i suoi figli da sinistra a destra, poi i figli dei suoi figli da sinistra a destra e così.

Questi ordini sono utilizzati in diversi algoritmi che coinvolgono gli alberi: si tratta di argomenti che non fanno parte di ProgIIB. Facciamo solo qualche cenno. Se vogliamo visitare un albero spostandosi da un nodo a un figlio oppure ritornando al nodo-padre possiamo adattare la visita in pre-ordine. Per disegnare un albero da destra a sinistra facciamo comparire i nodi in in-ordine. Per valutare una espressione algebrica valutiamo in post-ordine l'albero delle sue sotto-espressioni. Quando i rami di un albero rappresentano delle scelte in un gioco, utilizziamo una forma di esplorazione per livelli per scegliere una strategia di gioco.

```
//Tree.java

public abstract class Tree {
//Test se l'albero e' vuoto
public abstract boolean empty();

//stampa i nodi in pre-ordine: radice-sinistra-destra
    public abstract void preOrder();
//stampa i nodi in in-ordine: sinistra-radice-destra
    public abstract void inOrder();
//stampa i nodi in pre-ordine: sinistra-destra-radice
    public abstract void postOrder();
```

```
// Metodo che restituisce il livello numero n sotto forma di stringa
    public abstract String livello(int n);
// Metodo che stampa per livelli
    public abstract void livello();
// Metodo leavesAt(n) che calcola il numero di foglie
//di un albero a distanza n dalla radice, per n>=0
   public abstract int leavesAt(int n); //n>=0

protected abstract void scriviOutput
  (String prefix, String root, String left, String right);
//Metodo che gestisce la parte NON pubblica della stampa.
//Non forniamo spiegazioni sul suo funzionamento.

 public void scriviOutput()
  {scriviOutput("","___","   ","   ");}
// Metodo pubblico di stampa, dall'alto verso il basso, con i
// sottoalberi disegnati piu' a destra dell'albero di cui fan parte
}


//Leaf.java     unico elemento: "leaf" (albero vuoto)

public class Leaf extends Tree {
public boolean empty(){ return true; } //l'albero vuoto e' vuoto
public void inOrder() {}
public void preOrder() {}
public void postOrder(){}

public String livello(int n){return "";}
public void livello(){}

public int leavesAt(int n){if (n==0) return 1;  else return 0; }

protected void scriviOutput
 (String prefix, String root, String left, String right)
 { System.out.println(prefix + root + "leaf"); }}

public class Branch extends Tree {
 private int elem;   //radice
 private Tree left, right;

//Branch.java
```

```
public Branch(int elem, Tree left, Tree right)
 {this.elem = elem; this.left = left; this.right = right;}

public boolean empty(){ return false; }
// Un albero non vuoto non e' vuoto

/* visita in-order:scendo a sinistra,visito la radice, scendo a
destra */
public void inOrder()
 {left.inOrder();System.out.print(elem+ " ");right.inOrder();}

/* visita pre-order: visito la radice, scendo a sinistra, scendo a
destra */
public void preOrder()
{System.out.print(elem+" ");left.preOrder(); right.preOrder();}

/* visita post-order: scendo a sinistra, scendo a destra, visito la
radice */
public void postOrder()
{left.postOrder();right.postOrder();System.out.print(elem+" ");}

public String livello(int n)
 {if (n==0) return elem + " ";
  else return left.livello(n-1) + right.livello(n-1); }

 public void livello()
{int liv=0; String s = livello(liv);
  while (s.length() > 0)
   {System.out.println(s);liv++;s=livello(liv);}}

 public int leavesAt(int n)
 {if (n==0)   return 0;
  else  return left.leavesAt(n-1) + right.leavesAt(n-1);}

//Metodo che gestisce la parte NON pubblica della stampa.
protected void scriviOutput
(String prefix, String root, String left, String right)
{this.left.scriviOutput(prefix+left, "   /", "    ", "  ¦");
 System.out.println(prefix + root + "[" + elem + "]");
 this.right.scriviOutput(prefix+right, "   \\", "  ¦", "   ");}}
```

```
// TestTree
import java.util.*;

public class TestTree {public static void main(String[] args){

Tree t = new Branch(1,
                    new Branch(2,
                               new Leaf(),
                               new Leaf()),
                    new Branch(3,
                               new Leaf(),
                               new Branch(72,
                                          new Leaf(),
                                          new Branch(9,
                                                     new Leaf(),
                                                     new Leaf()))));

/* Albero t (radice a sinistra, figli dall'alto in basso)

           /leaf
         /[2]
        ¦  \leaf
 ______[1]
        ¦   /leaf
         \[3]
           ¦   /leaf
            \[72]
              ¦   /leaf
               \[9]
                  \leaf

Livello                  Albero t (radice in alto, figli da sinistra)
-------------------------------------------------|---------------
0                                                    ____1_____
-----------------------------------------------/----------\-----------
1                                                2           _3_
-----------------------------------------/\----------/---\--------
2                                      leaf leaf   leaf    72
-----------------------------------------------------------/--\-----
3                                                        leaf   9
-----------------------------------------------------------------/\---
4                                                           leaf leaf
```

```
t in pre-order:   1 2 3 72 9
t in in-order:    2 1 3 72 9
t in post-order:  2 9 72 3 1
t per livelli:    1 2 3 72 9              */

System.out.println( "\n L'albero t: \n");        t.scriviOutput();

System.out.println( "\n Visita pre-order t:");   t.preOrder();
System.out.println( "\n Visita in-order t:");    t.inOrder();
System.out.println( "\n Visita post-order t:");  t.postOrder();
System.out.println( "\n Visita per livelli t:"); t.livello();

System.out.println( " Foglie per livello t:");
 for(int i=0;i<=5;i++)
  System.out.println( "t.leavesAt(" + i + ") = " + t.leavesAt(i));
/* risultato: foglie per livello = 0 0 3 1 2 0 */
    }
}
```

Corso: "Programmazione IIB" Descrizione delle 24 Lezioni e delle 3 Esercitazioni

Università di Torino
Corso di Laurea in Informatica
A.A. 2018-2019

Le lezioni del corso di ProgIIB nel 2019 corrispondono ai capitoli 9-15 del libro di testo (Walter Savitch, *Programmazione di base e avanzata con Java)*, tranne per il capitolo 14, Stream, che viene **omesso**, e per il **capitolo 15, Strutture Dinamiche**, che viene **anticipato e ampliato**.

Rispetto al libro, dedichiamo più spazio a: ***(1)*** le definizioni ricorsive, ***(2)*** all'uso del comando ***assert*** per sollevare un'eccezione quando il programma non si comporta come previsto, ***(3)*** al disegno di Stack e Heap. Tutti e tre i punti corrispondono a un esercizio di esame.

Lezione 01. Venerdì 01/03/2019. Presentazione del Corso. Capitolo 8 del libro. Classi: attributi e metodi pubblici di un oggetto. Un esempio preso libro: la classi Cane.

Lezione 02. Martedì 05/03/2019. Capitolo 8 del libro. Attributi e metodi private, metodi get e set per un attributo. Un esempio preso dal libro: la classe Specie. Un esempio che rivedrete a Laboratorio: la classe Calcolatrice.

Lezioni 03. Venerdì 08/03/2019. Finiamo il capitolo 8, con le assegnazioni di oggetti e il metodi equals, e iniziamo il capitolo 9, con i costruttori di una classe.

Lezione 04. Martedì 12/03/2019. La classe "Stack" delle pile, che sarà usata in laboratorio. Un esempio di metodi che si richiamano l'un l'altro: la classe Oracolo.

Esercitazione 01. Mercoledì 13/03/2019. Esercizi su attributi e metodi, statici e dinamici, pubblici e privati: classi Matita e Elicottero.

Lezione 05. Venerdì 15/03/2019. Rappresentiamo l'indovinello sulle bottiglie da 3 e 5 galloni del film "Die Hard 3", usando due classi, Bottiglia e Die Hard. Descriviamo una classe con attributi e metodi privati e information hyding: la classe Frazione.

Lezione 06. Martedì 19/03/2019. Rubrica, il primo esempio di classe definita a partire dai vettori di oggetti di un'altra classe (capitolo 9.6), Contatto.

Lezione 07. Venerdì 22/03/2019. Capitolo 9.5: rappresentazione dell'associazione tra classi attraverso diagrammi UML. La classe ***ArrayExt*** che rappresenta i vettori estendibili.

Lezione 08. Martedì 26/03/2019. Un esempio di accesso indebito ai dati *(Security Leak)*: le classi CoppiaAnimale e Hacker. La classe Node (anticipazione del cap.15). La classe DynamicStack che rappresenta le pile estendibili.

Esercitazione 02. Mercoledì 27/03/2019. Una classe per le code dinamiche su interi.

Lezioni 09. Venerdì 29/03/2019. Esercizi: libreria di metodi statici con argomento nella classe Node.

Lezioni 10. Martedì 02/04/2019. Classi generiche: coppie, nodi e pile generiche.

Lezione 11. Venerdì 05/04/2019. Un primo di esempio di ereditarietà (cap. 10), l'estensione della classe Bottiglia, aggiungendo a ogni bottiglia due possibili stati: aperta/chiusa. Ripasso dell'uso di assert, con un esercizio di esame.

Lezione 12. Martedì 09/04/2019. Un esempio di estensioni ripetute di classi, aggiungendo alle pile dinamiche prima un attributo "massimo valore", poi un attributo "size".

Lezione 13. Giovedì 11/04/2019. Il concetto di tipo esatto di un oggetto (il più piccolo tipo per un certo oggetto, quello con cui l'oggetto viene definito). La scelta del metodo da applicare a un oggetto in base al suo tipo esatto (cap. 10).

Lezione 14. Martedì 16/04/2019. Un esempio di ereditarietà e uso del tipo esatto con la classe Persona e le sue sottoclassi. Una implementazione del tipo vettore attraverso linked lists (liste di nodi collegati da puntatori), con accesso in sola lettura all'interno della lista usando la classe MiniIterator.

Esercitazione 03. Mercoledì 17/04/2019. Riprendiamo il disegno di un vettore di figure in una finestra Java visto nella Lezione 13 e aggiungiamo nuove forme e nuovi parametri.

Lezione 15. Martedì 30/04/2019. Primo esempio di uso di classi astratte (capitolo 11 del libro di testo), attraverso una classe astratta Figure per il calcolo di area e perimetro di semplici figure geometriche.

Lezione 16. Venerdì 03/05/2019. Liste ordinate per rappresentare insiemi, definite come una classe astratta **Lista**.

Lezione 17. Martedì 07/05/2019. Implementazione degli alberi di ricerca su interi usando le classi astratte.

Lezione 18. Venerdì 10/05/2019. Interfaccie (capitolo 11.3). Generici vincolati. Classe astratta generica degli alberi di ricerca.

Lezione 19. Martedì 14/05/2019. Interfacce. Esempi: Comparable, Iterator e Iterable, implementazione di queste interfaccie nelle classi Bottiglia e ListExt.

Lezione 20. Venerdì 17/05/2019. Esempio di un compito di esame con la soluzione.

Lezione 21. Martedì 21/05/2019. Eccezioni controllate e non controllate: esempi. Una classa di Frazioni con tipo di eccezioni controllate: DenZeroException.

Lezione 22. Venerdì 24/05/2019. Prima ora: la classe Interval di intervalli in una lista come implementazione di Iterable<Integer>. Seconda ora: un analizzatore lessicale per espressioni composte da numeri naturali e le quattro operazioni, con eccezioni controllate di tipo IOException.

Lezione 23. Martedì 28/05/2019. Esempio di compito di esame risolto.

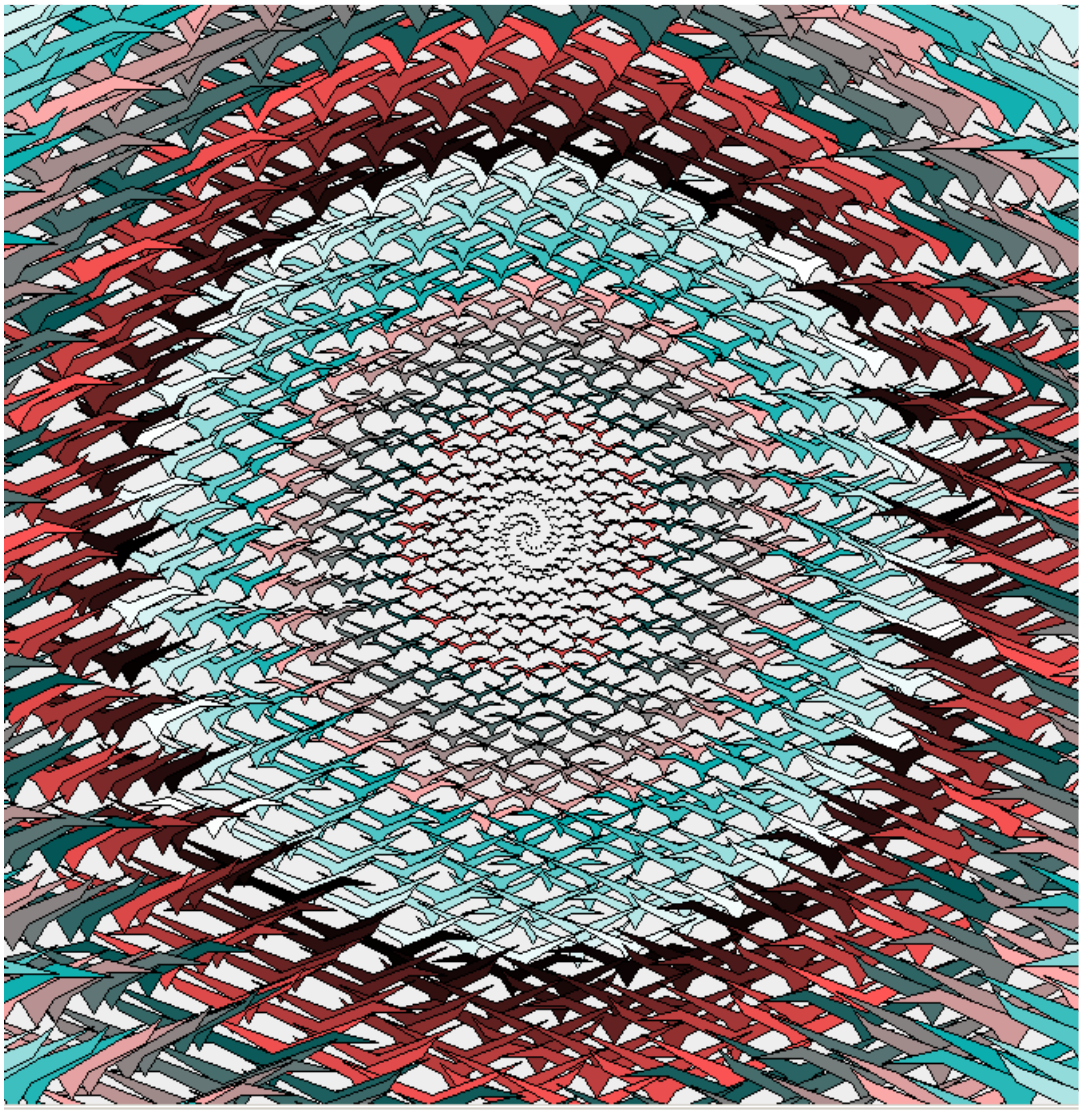
Lezioni 24. Venerdì 31/05/2019. Esercizi su alberi: visite in preordine, inordine, postordine, per livelli, foglie in un livello.

Con la Lezione 24 termina il corso di ProgIIB. *Notate che il capitolo 14, Stream, viene omesso, e che il capitolo 15, Strutture Dinamiche, è stato anticipato e ampliato.*

31/05/2019
Fine del Corso:

Programmazione IIB 2018/2019

Corso di Laurea in Informatica
Università di Torino



Una spirale disegnata con la classe Java "Figure" vista a lezione