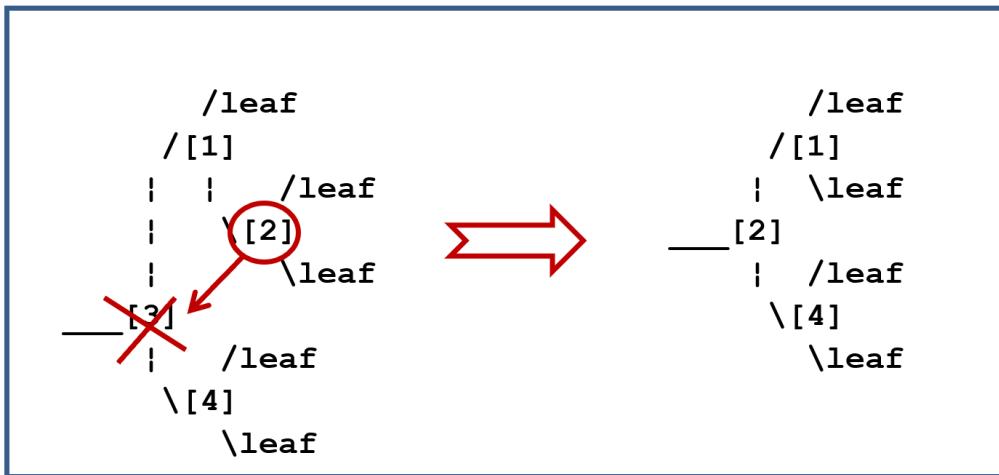


Corso: "Programmazione II"

Lezioni e Esercitazioni

Università di Torino
Corso di Laurea in Informatica
A.A. 2022-2023



Una immagine di un algoritmo che vedremo nel corso, per la rimozione di un nodo da un albero di ricerca

Autori delle Dispense

- Versione 2020: S. Berardi, materiale originario di L. Padovani e V. Bono, con contributi di: D. Magro, G. Torta, M. Baldoni, S. Tedeschi.
- Revisione 2021: V. Bono. Ringraziamo L. Padovani e C. Cattuto per le correzioni suggerite nel 2020.
- La versione 2021 è stata resa accessibile ai non vedenti (luglio 2021). Ringraziamo B. Lopardo e A. Albano per la collaborazione.
- Revisione 2022: V. Bono.
- Revisione 2023: S. Berardi, V. Bono. Con contributi di F. Ciravegna.

Presentazione del Corso: Programmazione II - Teoria

Nel 2022/2023 questo corso si svolge in 48 ore (24 lezioni di 2 ore) e vale 6 crediti, è associato a 30 ore di laboratorio (10 lezioni di 3 ore, su due turni, T1 e T2).

Il corso è la continuazione del corso di **Programmazione I** (ProgI) del 2022/2023: supporremo note le **dispense di ProgI**, che trovate per esempio su:

<http://www.di.unito.it/~stefano/21-09-20-Dispense-Programmazione-1-CorsoB.pdf>

Testo di consultazione

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

Programma del corso in breve

Rispetto al libro dedicheremo più spazio a: **1.** definizioni ricorsive **2.** ereditarietà. **3.** asserzioni **4.** disegno del modello della memoria di Java (*stack* e *heap*). **Tutti e 4 i punti potranno corrispondere a un esercizio di esame.**

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

Organizzazione e distribuzione del materiale didattico per Programmazione II A, B e C

L'organizzazione e il materiale di Programmazione II A, B, e C sono reperibili nei rispettivi siti I-learn dei corsi. Ci sono anche i siti I-learn corrispondenti ai vari turni di laboratorio.

In tutti i corsi di ProgII del 2022-22, il materiale didattico è distribuito come segue:

- Gli argomenti per il corso e il laboratorio sono di norma pubblicati sul sito I-learn corrispondente all'inizio di ogni settimana.

- La **versione definitiva del codice sorgente** visto a lezione è di norma pubblicato entro il termine di ogni settimana (in particolare, il codice viene anche **inserito in queste dispense**, potete copiarlo e incollarlo).
- 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 **anticipare o posticipare** la pubblicazione del codice per ragioni didattiche. In queste dispense di ProgII trovate **le versioni preliminari delle lezioni di ProgII non ancora svolte** con relativo codice. Attenzione però: questa parte è preliminare, quindi può cambiare (**anche se di solito in modo minimo**) fino alla pubblicazione definitiva, la settimana dopo la lezione stessa.

Un'ultima osservazione: il codice delle lezioni è **sovra-commentato** (fa parte della lezione), mentre lo stesso codice quando rivisto nel corso di laboratorio è **commentato solo nei punti essenziali**, come deve essere per un prodotto finito.

Ambiente di sviluppo e visualizzatore Java

Come ambiente di sviluppo (contenente anche il compilatore Java e la JVM) scegliete quello che preferite: per suggerimenti rimandiamo alle pagine dei laboratori di ProgI di quest'anno.

Vi consigliamo di usare un Java Visualizer per visualizzare l'organizzazione della memoria di semplici programmi, ovvero per avere una rappresentazione di stack e heap durante l'esecuzione del programma. Vi consigliamo, per esempio:

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

(**Attenzione:** in questo visualizzatore dovete **crocettare tutte le opzioni di visualizzazione che vi vengono date**, altrimenti la memoria Java viene rappresentata in modo troppo semplificato; inoltre dovete schiacciare il tasto **stdin** e inserire nella finestra di input le righe di input richieste dal programma **prima di iniziare la simulazione**).

All'esame vi potremmo chiedere di **disegnare stack e heap a mano** per un esempio.

Modalità di esame

Le modalità d'esame si trovano nella pagina ufficiale del corso di Programmazione II (vi si può accedere tramite il link presente nell'orario) e nelle pagine I-learn summenzionate.



L'icona di Java: una tazzina di caffé

Indice delle Lezioni e Esercitazioni di Programmazione del 2022/2023

Lezione 01	10
Classi, attributi e metodi pubblici e privati	10
Un primo esempio di definizione di classe: la classe Gatto.....	12
Eseguibilità di una classe.....	12
Alcuni cenni importanti sulla visibilità delle classi.....	13
Metodi dinamici pubblici.....	14
Variabili dinamiche.....	15
Lezione 02	25
Attributi e metodi privati, get e set	25
Lezione 02. Parte 1. Un primo esempio di attributi e metodi privati: la classe Specie	25
Lezione 02. Parte 2. Un esempio non banale di attributi e metodi privati: la classe Calcolatrice.....	30
Una calcolatrice semplificata.....	30
La calcolatrice come oggetto.....	30
Metodi e attributi privati.....	31
Lezione 03	36
Assegnazioni di oggetti, metodo equals, costruttori	36
Lezione 03. Parte 1	36
Uso di costruttori con lo stesso nome.....	36
Uguaglianza di oggetti.....	36
Lezione 03. Parte 2	40
Soluzione dell'esercizio proposto nella Lezione 02	45
Lezione 04	48
La classe "Stack", chiamate di metodi	48
Lezione 04. Parte 1. Un primo esempio di libreria: la classe Stack	48
Incapsulamento dei dati.....	49
Uso di "assert".....	49
Abilitazione delle asserzioni.....	49
Lezione 04. Parte 2. Chiamate tra metodi	52

Esercitazione 01	56
Attributi e metodi statici e dinamici	56
La classe Matita	56
La classe Elicottero	57
Lezione 05	63
Modelli di oggetti reali e Information Hiding	63
Lezione 05. Parte 1. Metodi statici e dinamici	63
« <i>Die Hard 3: the Water Jug Riddle</i>	63
Lezione 05. Parte 2. Un esempio di encapsulamento di dati: la classe Frazione	67
Altri possibili metodi per la classe Frazione	68
Proprietà invariante per la classe Frazione	68
Lezione 06	72
Classi di array di oggetti	72
La classe Contatto	72
La classe Rubrica	73
Lezione 07	81
Diagrammi UML e array estendibili	81
Lezione 07. Parte 1	81
Classi in UML	81
Lezione 07. Parte 2. Array estendibili	83
La classe ArrayExt	83
Spostamento degli elementi di un array	84
Lezione 08	89
Security Leak, le classi Node e DynamicStack	89
Lezione 08. Parte 1	89
Lezione 08. Parte 2. Pile dinamiche	93
La classe Node	93
Esercitazione 02	100
Code dinamiche	100
Diagramma UML per le code dinamiche	103
Soluzione Esercitazione 02 del 2021	105
Lezione 09	107
Metodi statici per la classe Node	107

Lezione 09. La classe NodeUtil (esercizi su liste concatenate)	107
Soluzioni esercizi 1-7 della Lezione 09	112
Lezione 10	116
Classi generiche: coppie, nodi e pile	116
Lezione 10. Parte 1: coppie generiche	116
Lezione 10. Parte 2: nodi generici e tipo Integer	119
Autoboxing: i tipi Integer, Boolean e Double.	119
Lezione 11	124
Ereditarietà e assert	124
Lezione 11. Parte 1. Un primo esempio di estensione di una classe: la classe BottigliaConTappo.	124
Estensione di una classe.	125
Lezione 11. Parte 2. "Assert o non assert, questo è il problema!" (di Luca Padovani)	128
Prima un breve ripasso sugli array di array.	132
Soluzione dell'esercizio 3 di esame (Lezione 11)	133
Lezione 12	135
Estensioni ripetute di classi	135
Un esempio di calcolo su una pila P di DynamicStackSize	140
Lezione 13	143
Tipo esatto e binding dinamico	143
Lezione 13. Parte 1. Tipo esatto e tipo apparente. Downcast e upcast, binding dinamico	143
Lezione 13. Parte 2. Un esempio di ereditarietà e di binding dinamico. 151	
Diagramma UML della classe Disegno	156
Lezione 14	158
Esempi di ereditarietà e array come liste	158
Lezione 14. Parte 1. Vediamo nuovi esempi di uso del dynamic binding.	158
Lezione 14. Parte 2. Le classi MiniLinkedList e Iterator.	162
Esercitazione 03	169
Array di figure	169
Soluzione Esercitazione 03	170
Esercitazione 03	173
Estensione facoltativa dell'esercizio	173

Lezione 15	174
Classi astratte di figure	174
Lezione 15. Parte 1. Una classe Figure per il calcolo di area e perimetro.	174
Regole per le classi astratte.	175
Lezione 16	185
La classe astratta Lista	185
Lezione 17	192
La classe astratta ricorsiva degli alberi di ricerca	192
Lezione 18	202
Generici vincolati, interfacce e alberi di ricerca	202
Interfacce Java.	202
L'interfaccia Comparable<T>.	202
Generici Vincolati.	203
Lezione 19	211
Interfacce Comparable, Iterator e Iterable	211
L'interfaccia Iterable<E>.	214
Lezione 20	221
Eccezioni controllate e non controllate	221
Eccezioni.	221
Eccezioni non controllate e controllate.	222
Cattura di eccezioni.	224
Come usare le eccezioni.	227
Esempi di eccezioni non controllate (cattura possibile)	227
Esempi di eccezioni controllate (cattura necessaria).	231
Lezione 21	234
Esempi di esercizi di esame con soluzioni	234
Lezione 21. Soluzioni degli esercizi assegnati	238
Lezione 22	246
Esempi di uso di Iterable e di eccezioni controllate	246
Lezione 23	253
Esempio di compito di esame	253
Lezione 24	266

Visita di un albero in pre-, in-, post-ordine e per livelli 266

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 01. Introduzione alla Programmazione a Oggetti.

Inizieremo la lezione con una introduzione alla programmazione ad oggetti e le sue principali proprietà di incapsulamento, ereditarietà, polimorfismo e astrazione. Ciascuno di queste proprietà sarà oggetto di altrettante (parti di) lezioni del corso.

La programmazione ad oggetti manipola - appunto - oggetti. Come esseri umani, noi siamo abituati a vivere in un mondo di oggetti che manipoliamo o da cui siamo manipolati. La nostra auto, per esempio è un oggetto che ha un certo numero di parti (il volante, i pedali, etc.) che a loro volta sono oggetti. Gli oggetti hanno funzioni e caratteristiche, come il colore o la funzione clacson del volante. Quando abbiamo bisogno di sapere delle informazioni dagli oggetti o di compiere azioni, accediamo alle loro qualità (per esempio leggiamo il numero della nostra carta di identità) o applichiamo una funzione (per esempio premiamo il clacson sul volante).

La programmazione ad oggetti riproduce questo paradigma di tutti i giorni nella programmazione. Gli oggetti che manipoliamo sono strutture software che spesso rappresentano oggetti della vita reale. Per esempio per stampare possiamo usare un oggetto di tipo stampante, che avrà certe caratteristiche (es. marca, identificativo univoco, etc.) e certe funzioni (e.g. stampa a colori o in bianco e nero a cui passo un file). L'oggetto software della stampante sarà ovviamente il suo proxy per operare nel mondo reale. Lo stesso accadrà per un robot. Ma altre volte (la maggior parte?) si tratterà di strutture dati come un database. Anche i database avrà un nome ed un indirizzo (caratteristiche) ed una serie di funzioni proprie (inserisci, distruggi, cerca, etc.).

Classi e istanze. E così come nel mondo reale anche nella programmazione abbiamo **classi di oggetti** che rappresentano l'astrazione dell'oggetto singolo (e.g. la classe automobile). E le

classi avranno sottoclassi che definiscono tipi specializzati (es. la classe automobile avrà come sottoclasse il modello Peugeot 208).

Le classi avranno istanze di oggetti (una Peugeot 208, una Fiat 500L eccetera). Quindi le classi definiscono il tipo astratto che ha specifiche caratteristiche (colore, targa) e funzioni (accensione, messa in marcia, etc.) a cui corrispondono azioni.

Le **istanze** stabiliscono l'oggetto individuale di tipo definito dalla classe. E così come nel mondo reale noi incontriamo solo oggetti individuali, i programmi possono solo manipolare istanze: così come io nel mondo reale non posso guidare il concetto generico di automobile, così il mio programma non potrà manipolare il concetto di quadrato. Io posso solo guidare una istanza di automobile, per esempio una Peugeot 208. Così un programma potrà solo manipolare le istanze delle classi di oggetti (e.g. un quadrato con uno specifico lato ed un preciso colore).

Vediamo un ulteriore esempio che vi sarà molto familiare nel mondo del software:

	A	B	C	D	E
1	Nome	Cognome	Indirizzo	Numero	Città
2	Giovanna	Rossi	via Gattinoni	12	Torino
3	Marcello	Vatte	via Vattelapescia	44	Canicattì
4	Antonietta	Digua	via Dilà	33	Milano

Nella programmazione ad oggetti, ogni riga tranne la prima è un oggetto. La classe è definita dalla prima riga, cioè dai nomi delle colonne che stabiliscono i nomi delle proprietà degli oggetti (in Java si stabilisce anche il tipo, cioè si dichiara che un certo campo può solo contenere ad esempio stringhe o solo numeri).

Quindi la classe definisce la Persona che avrà come attributi Nome, Cognome, etc. Ogni riga è un oggetto (istanza). Cioè c'è una persona con uno specifico nome, cognome, etc. Infine, la tabella è un array di oggetti, e a sua volta un oggetto più grande, dunque gli oggetti possono comparire dentro gli oggetti. Gli oggetti possono avere più forme. Se un oggetto rappresenta un auto, avrà colore, cilindrata, targa, etc, (cioè attributi di tipo diverso: colore ha tipo una classe Java, la targa è una stringa e la cilindrata è un numero).

Passiamo ora alla definizione formale di classe in Java.

Classi, attributi e metodi pubblici: definizione formale. Una classe è un costrutto di programmazione per creare oggetti. Un oggetto ha uno

stato, rappresentato dai valori contenuti in *attributi* (detti anche *campi*), e questo stato può essere modificato da operazioni dette *metodi*. La classe definisce attributi e metodi. Una istanza di una classe non può avere attributi e metodi che non siano definiti dalla classe stessa (né più né meno come il volante di una particolare Peugeot 208 non avrà un pulsante in più delle altre Peugeot 208).

Un oggetto è l'indirizzo di un gruppo di dati più semplici, che sono la rappresentazione in memoria degli attributi. Ogni classe contiene anche un oggetto degenero **null** privo di attributi e di informazioni, lo stesso per tutte le classi. Un esempio di oggetto non banale è un array di interi (ProgI, Cap.6). Come già sapete, un array di interi 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 Gatto. Un oggetto x di classe Gatto (un "gatto" x, dove x è il nome di una variabile di tipo Gatto) è composto di dati più semplici, gli attributi, **detti anche campi**, dell'oggetto. Nel caso della classe Gatto scegiamo come attributi: Nel caso della classe Gatto scegiamo come attributi:

- una stringa (il **nome**)
- una stringa (la **razza**)
- un intero (gli **anni**).

Nella memoria un oggetto è rappresentato dall'indirizzo del suo primo attributo.

Gli attributi di un gatto x si "referenziano" (si scrivono) nel codice con: x.nome, x.raffa, x.anni, e più in generale con x.attributo. **Creazione di istanze.** Gli oggetti di una classe si costruiscono con un comando **new** che avete già visto usare per vettori e matrici, e che definisce un nuovo oggetto come l'indirizzo di un'area di memoria attualmente non utilizzata, **che conterrà gli attributi dell'oggetto**. Il comando C x = new C(); definisce un nuovo oggetto x nella classe C, con valori di default per gli attributi.

Eseguibilità di una classe. Nel corso di ProgI 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 Gatto senza un main, allora Gatto **non è eseguibile** da sola, ma può venir usata dai metodi (e quindi anche da un main) **di** altre classi.

Alcuni cenni importanti sulla visibilità delle classi. In Java, le classi sono contenute in file e un file può contenere anche più di una classe. Possiamo dichiarare la classe Gatto, e una qualunque classe, **visibile** da parte delle altre classi del programma, in più modi. Vediamo ora le dichiarazioni più comuni: **(1)** Possiamo scrivere **class Gatto**: in questo caso Gatto è utilizzabile solo da classi che stanno in file che sono contenuti nella stessa cartella o **package** (in Java una cartella viene chiamata un *package*). **(2)** Possiamo scrivere invece **public class Gatto**: in questo caso la classe può essere usata da classi in altri file che stanno anche in altri package, previo l'utilizzo di istruzioni di *import*, o assegnazione delle variabili *d'ambiente*, o di opportuni parametri dati al compilatore. **La classe public è la forma più comune. Tuttavia, nel corso di ProgII, ci limiteremo (almeno all'inizio e salvo diverse indicazioni) alla seguente pratica:**

- Tutte le classi stanno in *un solo file*, inclusa **l'unica classe, per esempio chiamata C, che contiene il main**: il file deve avere nome **C.java**.
- **Solo la classe C** deve essere dichiarata **public**.

L'uso dell'oggetto NULL. Abbiamo detto che ogni classe contiene anche un oggetto degenero **null** privo di attributi e di informazioni, lo stesso per tutte le classi. A cosa serve definire un oggetto null? L'oggetto null rappresenta la mancanza: per esempio un certo rossi non ha un auto. Quindi se in una applicazione in cui le persone hanno un attributo automobilePosseduta (che verrà riempito con un oggetto di classe Automobile), se io cercassi di accedere alla variabile *rossi.automobile* il valore ritornato sarebbe null. Bisogna sempre controllare se un valore ritornato da un metodo o un attributo non è null. Se cerco di operare su di un oggetto di tipo null e per esempio accedere a una sua variabile o metodo, avrò un run time error. Per esempio:

```
Automobile auto = rossi.automobile
System.out.println(auto.colore)
```

Quando eseguirò riceverò:

```
>>> error Null Pointer Exception1
```

¹ Controllate sempre se un valore ritornato è null: le Null Pointer Exceptions sono il più comune errore nei programi Java e sono un autentico incubo da trovare perché non accadono a tempo di

Un'altra caratteristica di Object Oriented Programming: l'incapsulamento. Un oggetto ha metodi propri che sono delle funzioni (sequenze di istruzioni) che vengono eseguite dall'oggetto stesso, senza che il mondo esterno (gli altri oggetti o il main programme) vedano come questi vengano implementati.

Nel mondo reale noi tutti abbiamo questa esperienza: noi usiamo il volante della macchina (e.g. sterzando o suonando il clacson) senza sapere necessariamente come questo funzioni. I vantaggi dell'incapsulamento sono:

- Nasconde l'implementazione: l'implementatore della classe può cambiare e finchè il comportamento visibile dei metodi non cambia, l'utente esterno non deve cambiare il proprio comportamento. L'incapsulamento è la ragione per cui noi possiamo guidare indifferentemente una automobile diesel e una a benzina. Non importa come il motore è implementato: l'interfaccia esterna dell'oggetto (i suoi metodi di operazione) sono identici e quindi noi possiamo concentrarci sulla guida.
- Permette di proteggere le strutture interne da interferenze: per esempio due classi che definiscono due variabili o metodi con lo stesso nome non avranno problemi se le variabili sono interne (private) e inaccessibili al mondo esterno.

Nella descrizione qui sopra abbiamo definito l'interfaccia esterna (accessibile al mondo esterno) e le strutture interne all'oggetto.

Le prime sono rappresentate da attributi e metodi **public**, le seconde da attributi e metodi **private**.

Metodi dinamici pubblici. Nelle dispense di ProgI (Cap.3) avete visto i metodi **statici**, divisi in **pubblici** (utilizzabili fuori dalla classe in cui sono definiti) e **privati** (utilizzabili solo nella classe in cui

compilazione, ma di esecuzione: quindi non emergono fino a quando un utente arriva a un determinato punto del programma con certi valori. La cosa potrebbe non accadere a lungo.

Il problema dei null pointers è così grave che Android ha abbandonato Java in favore di Kotlin proprio per evitare gli errori di Null Pointer Exception che causavano problemi alle app. Infatti, linguaggi più moderni come Kotlin che costruiscono sull'esperienza Java, hanno metodi robusti per controllare già a tempo di compilazione se una variabile potrebbe assumere un valore null a tempo di esecuzione

sono definiti). Un metodo statico prende dei valori in ingresso e restituisce un valore in uscita oppure modifica la memoria. Una classe può anche contenere metodi **dinamici**. Un metodo dinamico (pubblico o privato) prende dei valori in ingresso e restituisce un valore in uscita oppure modifica la memoria, e inoltre viene chiamato "**facendolo interagire**" con un oggetto, e ci restituisce la risposta dell'oggetto. Cominciamo spiegando i metodi **dinamici pubblici**. Un metodo dinamico e pubblico si scrive:

```
public T metodo(argomenti){... istruzioni ...}
```

Non è necessario precisare che un metodo è dinamico: **qualunque metodo non dichiarato statico** si intende dinamico. Per ora supporremo che non ci siano argomenti, dunque che (argomenti) sia la lista vuota (). Un metodo dinamico viene chiamato mandandolo a un oggetto. Sia x un oggetto di classe (quindi tipo) Gatto, se scrivo **x.metodo()** mando il metodo dinamico "metodo" a un gatto 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.

In linea di principio, ogni metodo dinamico ha una versione statica ottenuta spostando l'oggetto x a cui viene mandato il metodo tra gli argomenti del metodo: **x.metodo()** diventa **metodo2(Gatto x)**. Usiamo la versione dinamica di un metodo per **leggibilità**: scrivendo **x.metodo()** sottolineiamo che il compito principale del metodo è di interagire con l'oggetto x.

All'interno della definizione di "**metodo**", quando scrivo "**nome**", "**razza**", "**anni**" intendo nome, razza ed anni del gatto x. All'interno di metodo(), il gatto 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 gatto x a cui mando il metodo. Volendo "this" si può omettere, se non si producono ambiguità di identificatori: "**nome**" abbrevia "this.nome".

Variabili statiche e dinamiche. Dentro una classe possiamo dichiarare variabili statiche: queste rappresentano un valore che possiamo assegnare e riassegnare, e sono le variabili che avete conosciuto a Programmazione I. Invece i nomi degli attributi vengono detti variabili dinamiche. Attributi o variabili dinamiche sono tutte le variabili dichiarate dentro una classe e **non dichiarate statiche**. Indicano gli

attributi di ogni oggetto della classe: per un gatto sono nome, razza anni. A differenza delle variabili statiche che hanno un valore solo in ogni momento della computazione, le variabili dinamiche hanno **un valore per ogni oggetto della classe**. Vengono usate proprio tramite un oggetto specifico della classe: se `x` è un nome di oggetto di tipo Gatto e "nome" una variabile dinamica della classe Gatto (un attributo della classe Gatto) allora come abbiamo visto `x.nome` è il nome di `x`. Per due gatti diversi `x`, `y` i valori di `x.nome`, `y.nome` saranno diversi. Le variabili dinamiche sono anche dette **campi**. I valori delle variabili dinamiche, ovvero degli attributi, rappresentano lo **stato** dell'oggetto. In una variabile statica, invece, possiamo mettere delle informazioni comuni a tutti gli oggetti della classe, per esempio il numero delle zampe dei gatti, che vale 4 per tutti i gatti.

Un esempio di metodo dinamico: i metodi di input. Come esempio, utilizziamo una classe **Scanner**. Uno oggetto 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 scriviamo `x.nextLine()`, allora inviamo il metodo `nextLine()` a un oggetto di tipo Scanner `x`, e se inseriamo una stringa da tastiera otteniamo in risposta la rappresentazione Java della stringa.

Un altro esempio di metodo dinamico: il metodo `toString()`. Ogni classe definita in Java ha un metodo dinamico **String `toString()`**. Se `x` è un oggetto di una classe `C`, allora se scrivo `x.toString()` ottengo la stringa di caratteri: "`C@indirizzo_esadecimale_x`". Questa stringa identifica univocamente `x`, ma per il resto contiene informazioni banali (classe e indirizzo di memoria di `x`). Quando faccio stampare `x` con `System.out.println(x)`; ottengo la stampa di `x.toString()`. Se invece ridefiniamo il metodo `toString()`, come vedremo, possiamo dare in forma leggibile delle informazioni significative su `x`.

Infine una discussione: in Java dobbiamo preferire i metodi statici o dinamici? In Java e nella programmazione ad oggetti in generale ci si focalizza sui **metodi dinamici**. Un metodo dinamico (pubblico o privato) prende dei valori in ingresso e restituisce un valore in uscita oppure modifica la memoria, e inoltre viene chiamato "mandandolo" a un oggetto, e ci restituisce la risposta dell'oggetto. Esempi di metodi dinamici nel mondo reale sono per esempio il suonare il clacson sul volante. In questo caso, il richiamo del metodo dinamico si scrive per esempio `x.suonaClacson()`. Preferiamo non passare

l'oggetto come parametro come si farebbe per esempio con un metodo statico come `sum(x, y)` per la somma di due valori: il metodo `suonaClacson()` è come fosse associato all'oggetto. È un po' come dire "sul volante, suona il clacson".

I metodi e gli attributi statici sono usati in linguaggi come il C ma vengono **evitati in programmazione a oggetti** ogni volta che è possibile. Evitare variabili e metodi statici quando si possono definire metodi dinamici (cioè associati all'oggetto).

Un solo metodo statico è necessario: il **main**. Se ne aggiungete un altro dovete avere una spiegazione robusta sul perché non fosse implementabile un metodo dinamico. Ugualmente, tutti gli attributi sono di regola dinamici, lasciando solo come statiche le caratteristiche che sono attribuibili alla classe di per se invece che all'individuo. Esempio di queste variabili statiche sono le costanti invarianti (e.g. il numero di zampe di un gatto non è una caratteristica che cambia da individuo a individuo, quindi è meglio che sia statico perché viene definito una volta per tutte nella classe). Vedremo meglio nelle prossime settimane.

Nota sui caratteri ASCII. Quando scriviamo del codice Java tendiamo a non utilizzare gli accenti, perché alcuni compilatori non li leggono. Preferiamo usare al loro posto gli apostrofi: scriviamo `a'`, `e'`, `e'`, `i'`, `o'`, `u'`, `E'` al posto di `à`, `è`, `é`, `ì`, `ò`, `ù`, `È`. Per la stessa ragione scriviamo `''` e `""` al posto di `''` e di `""`, che alcuni compilatori non leggono.

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

class Gatto { /* Le classi iniziano con la maiuscola */
    // Classe come generatore/"blueprint" di oggetti

    // Stato: campi dell'oggetto (a loro volta possono essere degli
    // oggetti)

    /** Ogni classe induce un tipo oggetto che corrisponde alla classe
    stessa. Per esempio, la classe Gatto induce il tipo Gatto, che può
    essere usato come qualsiasi altro tipo predefinito Java: come se fosse
    int, double, boolean.
```

```

*/
```

```

public String nome;
public String razza;
public int anni;
/** Le operazioni sugli oggetti della classe sono rappresentate come
metodi dinamici pubblici: se senza argomenti, si scrivono:
    public tipo metodo(){... ...}
Variabili e metodi iniziano con la minuscola, se uniamo piu' parole
dalla seconda in poi iniziano con la maiuscola. Es.: leggiInput()
Dentro il metodo indichiamo con this l'oggetto di tipo Gatto a cui
applicheremo il metodo. */
```

```

private static 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 applicato a
"tastiera" richiede una riga di input da tastiera e restituisce una
stringa. */
```

```

public void leggiInput() {
//metodo dinamico: chiamandolo, chiediamo a un oggetto di tipo Gatto
//di assumere nome, razza e eta' che gli inviamo da tastiera
    System.out.println( " nome = " );
    this.nome = tastiera.nextLine(); //nome gatto che riceve il metodo
    System.out.println( " razza = " );
    this.razza = tastiera.nextLine(); //razza gatto che riceve il
metodo
    System.out.println( " anni = " );
    this.anni = tastiera.nextInt(); //eta' gatto che riceve il metodo
    tastiera.nextLine();/* consumo il return dopo il numero anni */
/* si "consuma" il carattere return dopo l'ultimo dato inserito,
in questo caso il numero degli anni */

public String toString(){
/* metodo dinamico: chiamandolo, chiediamo a un oggetto di tipo
Gatto di fornire una stringa contenente i suoi dati. Possiamo
abbreviare this.nome con nome, eccetera. Ogni "\n" inserisce un a
capo. */
    return " nome = " + nome + "\n razza = " + razza +
           "\n anni = " + anni;
}
```

```

public int getEtaInAnniUmani(){
//metodo dinamico: chiamandolo, chiediamo a un oggetto di tipo
//Gatto di mandarci i suoi anni trasformati in anni corrispondenti
//per l'uomo. Conto 11 anni ciascuno i primi due anni del gatto,
//conto 5 anni ogni altro anno
    if (anni <=2)
        return anni*11;
    else
        return 22 + (anni-2)*5;
}
}

/** Ecco il primo esempio di un programma che usa classi definite da
noi: la classe GattoDemo ha un main, quindi e' un programma, e usa la
classe Gatto. La classe Gatto deve: (1) trovarsi nello stesso file del
programma (e' la nostra scelta), oppure (2) trovarsi in un file di nome
Gatto.java della stessa cartella ed essere public.
La classe GattoDemo deve essere salvata nel file GattoDemo.java
Per assegnare un attributo pubblico dell'oggetto Gatto x, per esempio
assegnare "anni" a 8 devo scrivere: x.anni = 8 */

public class GattoDemo {
//Una classe e' eseguibile se ha un main, come questo:
    public static void main(String[] args) {
        Gatto tramot = new Gatto();
        /** 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 gatto: null,
        null, 0 per gli attributi: nome, razza, anni */

        System.out.println( "tramot prima inserimento dati" );
        // stampo i valori di default: null, null, 0
        // se c'e' la toString() in Gatto
        System.out.println(tramot);
        System.out.println( "Inserisci dati tramot" );
        tramot.leggiInput();
        System.out.println( "Dati inseriti tramot" );
        // stampo i valori
        System.out.println(tramot);
        //qui "tramot" abbrevia "tramot.toString()"
        System.out.println( "eta' tramot in anni umani "
                            + tramot.getEtaInAnniUmani());
    }
}

```

```
Gatto galileo = new Gatto();
/** Questo crea un nuovo oggetto di tipo Gatto con valori di
 default null, null, 0 per gli attributi nome, razza, anni */
System.out.println("Inserisco dati galileo dentro il programma");
galileo.nome = "Galileo";
galileo.razza = "Persiano";
galileo.anni = 5;
System.out.println(galileo);
//qui "galileo" abbrevia "galileo.toString()"

}

}
```

/* Fornendo come input:

```
Tramot
Soriano
8
```

Otteniamo come output:

tramot prima inserimento dati

```
nome = null
razza = null
anni = 0
```

Inserisci dati tramot

```
nome = Tramot
razza = Soriano
anni = 8
```

Dati inseriti tramot

```
nome = Tramot
razza = Soriano
anni = 8
```

eta' tramot in anni umani 52

Inserisco i dati galileo

```
nome = Galileo
razza = Persiano
anni = 5 */
```

Includiamo ora la situazione della memoria a fine dell'esecuzione di GattoDemo. Potete ottenerla con un visualizzatore, per esempio:

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

Questo visualizzatore vi richiede il programma in un unico file, dunque dovete scrivere Gatto come classe senza il modificatore "public", seguita dalla classe GattoDemo come classe "public". (Ricordatevi di crocettare tutte le opzioni di visualizzazione che vi vengono date, altrimenti la memoria viene rappresentata in modo troppo semplificato; e di schiacciare stdin e inserire le righe di input **Tramot, Soriano, 8 prima di cominciare.**)

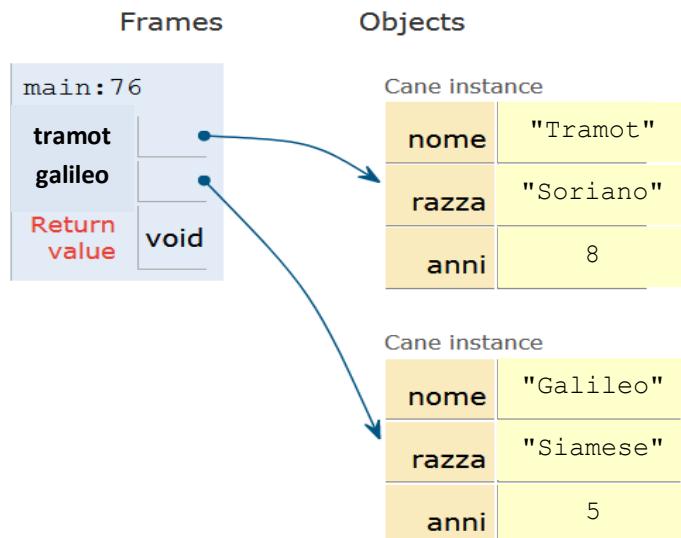
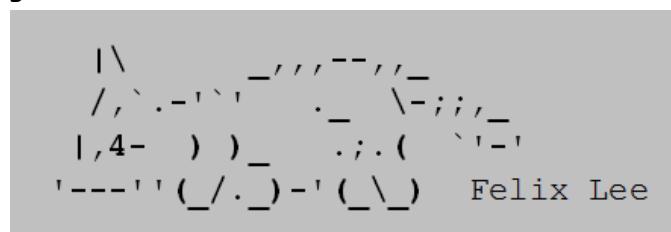


Diagramma Stack + Heap per l'esecuzione di GattoDemo. Vediamo uno stack che contiene le variabili **tramot** e **galileo**, da cui partono frecce che puntano nella Heap agli oggetti corrispondenti della classe Gatto. L'oggetto **tramot** contiene in questo istante del calcolo gli attributi **"Tramot"**, **"Soriano"**, **8**. L'oggetto **galileo** **contiene** gli attributi **"Galileo"**, **"Persiano"**, **5**.

Un disegno di un gatto con soli caratteri ascii (Felix Lee)



Nota tecnica sulla variabile riservata 'this'

La variabile 'this' è una variabile riservata di Java presente in ogni metodo dinamico e che serve per contenere l'indirizzo dell'oggetto su cui è stato invocato il metodo stesso (per esempio nel main() o in altro metodo). Tramite 'this', posso accedere a tutti i campi (e a tutti i metodi) dell'oggetto. Per esempio, se nel main() scrivo:

```
Gatto tramot = new Gatto();  
  
...  
  
tramot.leggiInput();
```

il 'this' in leggiInput() per questa chiamata assumerà il valore della variabile 'tramot' (di tipo Gatto) e quindi tramite essa potrò accedere ai campi dell'oggetto puntato (anche) da 'tramot'. Possiamo quindi dire che 'this' è un alias di 'tramot' durante l'esecuzione del metodo leggiInput().

Se invece nel main() scriviamo:

```
Gatto cipria = new Gatto();  
  
...  
  
cipria.leggiInput();
```

nell'esecuzione di leggiInput() che corrisponde a questa chiamata il 'this' diventerà l'alias dell'oggetto puntato dalla variabile 'cipria'.

La variabile 'this' si usa per referenziare i campi dell'oggetto, per esempio:

```
this.nome = "galileo";
```

Il 'this' può essere omesso, se non ci sono variabili locali o parametri che si possano confondere con "nome". Si può quindi scrivere:

```
nome = "galileo";
```

Diciamo che il "this" è "sintatticamente隐式": vogliamo dire che il 'this' è comunque presente anche se non è scritto esplicitamente.

Nota tecnica sul metodo `toString()`

Come detto più sopra, `toString()` è un metodo dinamico, ovvero viene invocato su un oggetto e quindi contiene la variabile implicita 'this', come spiegato sopra. Restituisce una stringa e di solito si usa per avere un "pretty printing" dell'oggetto su cui è invocato. Il metodo `toString()` con tipo di ritorno `String` e nessun parametro esplicito (i parametri tra parentesi) appartiene allo standard Java e in particolare è presente nella classe 'Object', e di conseguenza in tutte le classi Java. Di default stampa in esadecimale l'indirizzo dell'oggetto, ma può venir riscritta dal programmatore. Parleremo della gerarchia di classi e delle nozioni collegate di "ereditarietà" e override/sovrascrittura dei metodi più avanti.

IMPORTANTE: ricordatevi che il metodo `toString()` NON STAMPA! Restituisce una stringa che può essere utilizzata da un metodo che stampa, metodo che è bene non sia scelto a priori da Java, per lasciare maggiore libertà ai programmatore. Per esempio, quando passiamo un oggetto al metodo di stampa `System.out.println()`, come in:

```
System.out.println(tramot);
```

viene proprio richiamato il metodo '`toString()`' per trasformare "tramot" nella stringa "`tramot.toString()`", che viene poi stampata.

Se la classe dell'oggetto, in questo caso 'Gatto', contiene una sua versione di `toString()`, allora verrà eseguita questa, altrimenti verrà eseguita la `toString()` di 'Object', che restituisce la versione esadecimale dell'indirizzo di memoria heap dove è memorizzato l'oggetto. La `println()` stampa la stringa che la `toString()` le restituisce.

Per questo, se proviamo a commentare la definizione `toString()` in Gatto, otteniamo l'indirizzo dell'oggetto: infatti tolta la nostra definizione, resta la `toString()` definita nella classe Object che stampa l'indirizzo dell'oggetto. Invece, se de-commentiamo `toString()` in Gatto, otteniamo di nuovo una stampa dei campi concatenati dell'oggetto: infatti questa è la definizione che abbiamo scelto per `toString()` nella classe Gatto.

Vediamo ora alcuni esempi possibili di definizione di `toString()` in Gatto.

```
public String toString(){// versione scelta della toString():
```

```
return " nome = " + this.nome + "\n razza = " + this.razza +
"\n anni = " + this.anni; }

// una seconda versione:

//String s = " nome : " + nome;

//return s;

// una terza versione (abbastanza inutile, ma possibile):

// return "ciao";

// una quarta versione:

// return this.razza;

// una quinta versione - QUESTA E' SBAGLIATA!

// return System.out.println(razza);

/** SI DEVE RESTITUIRE UN RISULTATO DI TIPO String, mentre println
restituisce un valore di tipo "void"! INOLTRE NON E' COMPITO DI
toString() di stampare, ma di chi usa toString()!!! */
```

Lezione 02

Attributi e metodi privati, get e set

Lezione 02. 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 attributi di una classe. Tali metodi prendono il nome di **metodi get** (per i metodi che leggono l'attributo di un oggetto) e **metodi set** (per i metodi che scrivono l'attributo 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 pubblici rispetto a rendere pubblici gli attributi della classe è controllare in modo più fine le possibilità di accesso (es., omettendo il metodo set si impone il fatto che un attributo sia disponibile solo in lettura) e di impedire modifiche che producono dati contraddittori. Per evitare l'inserimento di dati contraddittori, di solito gli attributi di una classe sono privati. Per esempio, nel metodo *setSpecie()*, si imporrà che il valore della popolazione (campo 'popolazione') sia positivo.

Un attributo viene reso privato mettendo al posto del modificatore di visibilità **public** il modificatore **private**. Come primo esempio definiamo una classe **Specie** con metodi get e set (la trovate sul Savich al Cap. 8). Usando la visualizzazione della memoria, 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 eseguibile per rappresentare delle
specie di esseri viventi. Scrivereemo un programma per sperimentare
Specie in una classe di nome SpecieDemo, e salveremo tutto nel file:
SpecieDemo.java */
class Specie {
    /** Classe non pubblica, la mettiamo 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 (es. che non si accettino certi valori). */
public void setSpecie(String n, int p, double t){
    // se il parametro si chiamasse 'nome' invece di 'n',
    // il this esplicito sarebbe obbligatorio per non confondere
    // tra i due:      this.nome = nome;
    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". Se un dato e' riservato, basta togliere il suo metodo
"get" e l'attributo non e' piu' accessible dall'esterno della classe.
*/
public String getNome()           {return nome;}
public int     getPopolazione()   {return popolazione;}
public double  getTassoCrescita() {return tassoCrescita;}

private static Scanner tastiera = new Scanner(System.in);

public void leggiInput(){
    System.out.println( " nome = " );
    nome = tastiera.nextLine();

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

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

public String toString() //Stringa che descrive una specie
    {return " nome = " + nome + "\n popolazione = " + popolazione
+" \n 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 dice al compilatore che il reale p va visto come
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. Usando il Java Visualizer, vediamo
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 \n" +
                            bufaloTerrestre);
//bufaloTerrestre" abbrevia "bufaloTerrestre.toString()

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

        System.out.println( "Dati inseriti BufaloTerrestre \n" +
                            bufaloTerrestre);
//bufaloTerrestre" abbrevia "bufaloTerrestre.toString()

        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 dichiarati 'private':
        bufaloKlingon.nome = "BK"; //ERRORE! Dobbiamo scrivere, invece:*/
    }
}

```

```

bufaloKlingon.setSpecie("BK",1000,10);

System.out.println( "Dati inseriti BufaloKlingon \n" +
                     bufaloKlingon);
System.out.println( "Bufalo Klingon dopo 10 anni = "
                     + bufaloKlingon.prediciPopolazione(10));

System.out.println( "Identifico Bufalo terrestre e Klingon" );
bufaloTerrestre = bufaloKlingon;
// adesso le due variabili puntano allo stesso oggetto,
// come si vede stampadole:
System.out.println(bufaloTerrestre);
System.out.println(bufaloKlingon);
}

}

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

```

1. L'indirizzo di bufaloTerrestre diventa uguale a quello di bufaloKlingon. In altre parole, bufaloTerrestre adesso e' un alias di bufaloKlingon. Avete visto lo stesso fenomeno quando si assegna un array a un altro (ProgI sezione 6.4)
2. Abbiamo l'impressione che i dati del bufaloTerrestre 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 (tramite un processo che gira in parallelo con il nostro programma, detto Garbage Collector). */

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

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

(Ricordatevi di crocettare le opzioni e di schiacciare il tasto `stdin` e inserire gli input). Questo visualizzatore vi richiede il programma in un unico file, dunque dovete definire Specie come classe non pubblica.

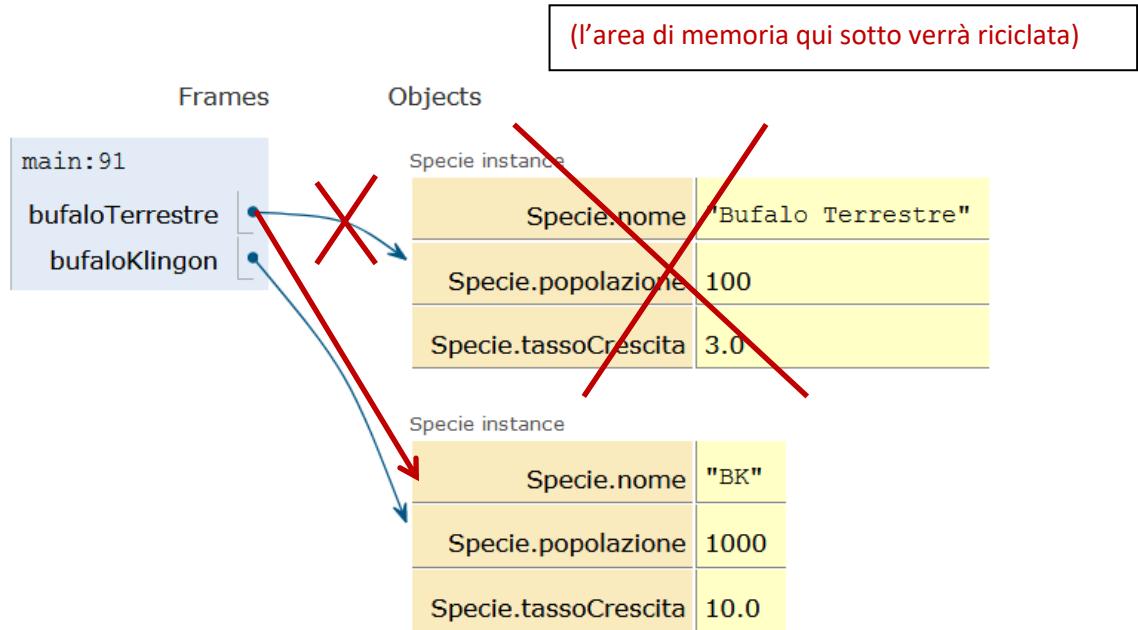


Diagramma stack + heap alla fine dell'esecuzione SpeciesDemo. Le variabili "bufaloTerrestre" e "bufaloKlingon" nel main puntavano ciascuna a una sua area di memoria. Dopo aver assegnato "bufaloTerrestre" a "bufaloKlingon" le due variabili puntano **alla stessa area**. L'area con le informazioni sul bufalo terrestre non è raggiunta da nessun puntatore e verrà **riciclata**

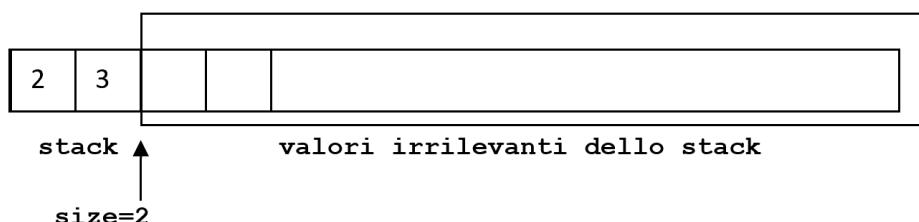
Vi invitiamo a fare il seguente esercizio, che spiega perché è bene che gli attributi di una classe siano privati. Definite una classe **Rettangolo** con attributi: **base**, **altezza e area**. Un rettangolo è correttamente definito se l'area è uguale a base per altezza. Accedendo dall'esterno posso 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 02. 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 oggetti 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, inseriti 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. Un oggetto calcolatrice deve avere un array **stack** di interi per memorizzare i risultati dei calcoli precedenti ancora da utilizzare, e un indice **size** per indicare quanti risultati ci sono in tale stack. Il motivo per cui l'array debba essere uno stack, ovvero una struttura LIFO (Last-In-First-Out), si capirà meglio nel seguito. Abbiamo bisogno di un metodo dinamico **int pop()** che restituisca l'ultimo risultato inserito nella calcolatrice, cancellandolo da stack, aggiornando size, e un metodo dinamico **void push(int x)** che aggiunga un valore x all'array stack, sempre aggiornando size. Infine ci vuole 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.

Un esempio. Supponiamo di aver appena inserito con due "push": push(2), push(3) i numeri 2 e 3 nello stack, portando il "size" dello stack a 2. Gli elementi del vettore dalla posizione 2 in poi sono irrilevanti, come indicato nel disegno qui sotto:



Supponiamo di voler eseguire una moltiplicazione tra gli ultimi due valori inseriti: allora eseguiamo due comandi "pop": riprendiamo dalla memoria 2 e 3, portando il "size" dello stack a 0. In realtà, il 2 e il 3 si trovano ancora sullo stack, ma dato che size vale 0 vengono considerati irrilevanti.

2	3				
---	---	--	--	--	--

↑ **Tutti i valori dello stack sono irrilevanti (2 e 3 hanno terminato la loro esistenza utile)**

size=0

Ora la calcolatrice moltiplica 2 e 3 e ottiene 6. In questo caso è stato inutile inserire e poi prelevare 2 e 3 dallo stack, in generale, quando i passi di calcolo da fare sono tanti, è necessario.

Metodi e attributi privati. Dobbiamo ora tradurre l'idea appena vista in metodi della classe Calcolatrice, scegliendo quali fare pubblici e quali 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 programmatore che non ne conoscono il funzionamento in dettaglio. Renderemo esegui **pubblico** e tutti gli altri metodi e attributi **privati**.

Una **descrizione alternativa della calcolatrice** sarebbe rimpiazzare gli attributi privati stack e size della classe Calcolatrice con **variabili stack e size poste nel main**. Se provate a scrivere una soluzione del genere, tuttavia, vi renderete conto che è più macchinosa. È molto più semplice nascondere stack e size nella classe Calcolatrice che renderli visibili solo al programmatore che deve implementare una calcolatrice.

Supporremo che le istruzioni di una calcolatrice siano scritte in **RPN** (**Reverse Polish Notation**, notazione inversa polacca), in cui le operazioni +, * seguono i loro argomenti anziché comparire tra il primo e il secondo argomento. Per esempio scriviamo "(2+3)*9" come "23+9*": il caratteri "23+" indicano (2+3) e aggiungendo "9+" otteniamo (2+3)*9. La notazione RPN semplifica, come vedremo, il lavoro alla calcolatrice, ma è difficile da leggere per un essere umano. Una calcolatrice più

realistica avrebbe anche un metodo per tradurre le istruzioni " $(2+3)*9$ " nella forma equivalente **RPN**: "23+9*". Per brevità noi non aggiungeremo questo metodo, che pure sarebbe importante avere.

Fate qualche prova di esecuzioni di istruzioni con il Java Visualizer. Il programma che trovate in Calcolatrice e CalcolatriceDemo è troppo lungo per essere simulato nel Visualizer, ma potrete fare eseguire singoli esempi.

```
//Salviamo il tutto nel file CalcolatriceDemo.java
class Calcolatrice {
    /** 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;

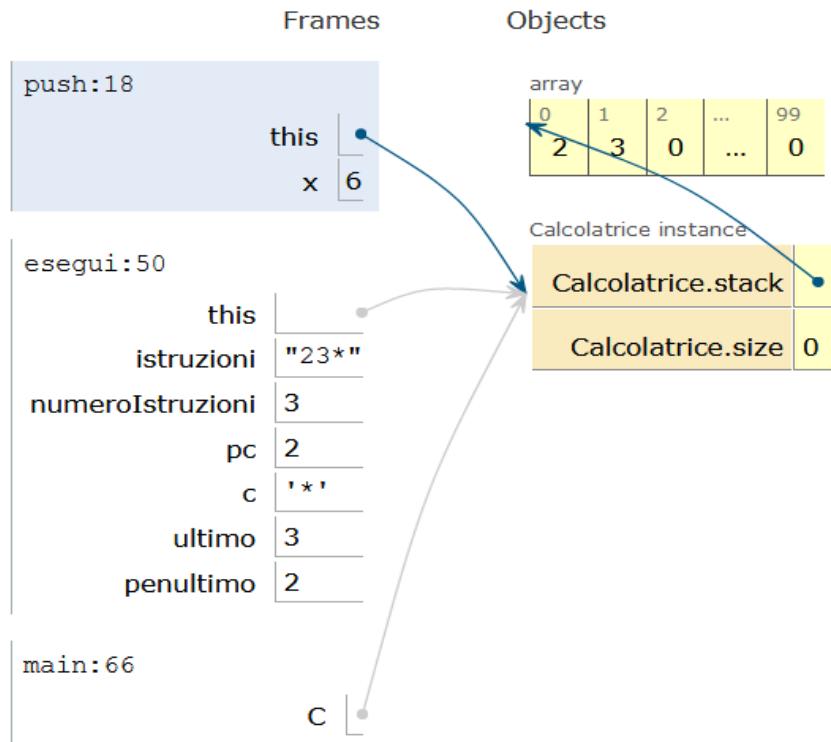
    /** push(x): aggiunge un intero x a stack dopo la parte utilizzata
     * e aumenta la parte di stack utilizzata di uno. */
    private void push(int x)
        {stack[size] = x; size++;}

    /** 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 delle istruzioni restituisco l'ultimo risultato
//ottenuto
}
}
```



Lo stato della memoria verso la fine del calcolo di $2*3$ nella calcolatrice C.
 Abbiamo già calcolato $6=2*3$, lo stack della calcolatrice contiene zero valori. Ora stiamo inserendo il risultato 6 nello stack con il comando `this.push(6)`. `this` indica l'indirizzo dell'oggetto "calcolatrice C"

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

}

}
```

Vediamo ora due esempi di errori di input per la calcolatrice:

- 1234455+: il risultato è 10 e non si ha errore a runtime, ma lo stack non risulta vuoto, ma contiene 12344;
- 1+++*: si ha una **ArrayOutOfBounds exception**, prodotto dalla ricerca del secondo argomento per la prima somma, argomento che viene cercato in posizione -1.

Lezione 03

Assegnazioni di oggetti, metodo `equals`, costruttori

Lezione 03. Parte 1 (30 minuti). Scrivendo `new C()` costruiamo un nuovo oggetto nella classe C, con tutti gli attributi inizializzati con valori di default (i valori di default dipendono dal tipo dei attributi, per esempio 0 per gli int): `new` alloca spazio nella heap e `C()` è l'invocazione a un metodo speciale, detto *costruttore*, che inizializza i attributi ai valori di default. In questo caso, viene invocato il costruttore detto di default, implicito nella classe C. In alternativa, possiamo definire noi stessi *dei costruttori pubblici* con la dichiarazione

```
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 assegniamo dei valori agli attributi della classe, utilizzando gli argomenti del metodo: per esempio, questi possono essere gli attributi del nuovo oggetto che stiamo costruendo. Se definiamo dei costruttori noi stessi, anche con zero parametri, il costruttore di default non è più accessibile.

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. I costruttori sono quindi *overloaded*, ovvero rappresentano operazioni con lo stesso nome, selezionate dal compilatore in base al tipo e numero dei loro parametri.

Uguaglianza di oggetti. Per decidere se due oggetti sono uguali, la prima soluzione è usare il test `x==y` di egualianza. Questo test controlla se `x,y` hanno lo stesso indirizzo, quindi se occupano la **stessa area di memoria**. Tuttavia, questo è più di essere uguali, significa che `x,y` sono due **alias** (nomi alternativi) per lo stesso oggetto. Per esempio, se `x, y` sono due array con gli stessi elementi

nello stesso ordine, ma posti in aree diverse di memoria, allora `x`, `y` risultano diversi se confrontati con `x==y`.

Un'altra possibilità è usare il metodo dinamico `equals` (Cap.8.3 del Savich). Se mandiamo a un oggetto `x` un metodo `equals(y)` con parametro un oggetto `y` otteniamo come risposta `x.equals(y) = true` o `false`, a seconda se `x` è "uguale" a `y` oppure no. Il metodo `equals` esiste in tutte le classi Java, ma nella maggior parte dei casi `x.equals(y)` viene definito in modo banale, come `x==y`. Tuttavia, **noi possiamo ridefinire equals**, e decidere quando consideriamo **due oggetti di una classe C uguali**: in genere controlliamo che tutti gli attributi siano uguali, ma questa non è l'unica soluzione.

Avete visto il problema di stabilire se due array sono uguali nelle dispense di ProgI (Sezioni 6.3 e 6.4): il problema di stabilire se due oggetti sono uguali è simile, eccetto che ora, se vogliamo, ridefiniamo il metodo dinamico `equals` in ciascuna classe che implementiamo.

Come esempio, vediamo un esempio del libro, una classe **`Animal`** per rappresentare gli animali: forniamo **(i)** due costruttori, uno con valori di default (ma esplicito) 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'egualanza 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
    /* Introduciamo una classe per sperimentare costruttori e metodo
equals. Gli attributi sono private. */
    private String nome;
    private int eta;
    private double peso;

    /** (i) Il primo costruttore assegna valori di default privi di
interesse (ma scelti dal programmatore) */
    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 conversione animale --> stringa */
public String toString()
    {return " nome " + nome + "\n eta' " + eta + "\n 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 è diverso dall'assegnare
direttamente x = y.
Con x = y: x e y occupano lo stesso spazio di memoria, sono lo
stesso oggetto e ogni modifica fatta a x si ripercuote su y. */

/** (iv) Metodo equals che controlla se due animali hanno gli stessi
valori di attributi. Uso il metodo dinamico s.equalsIgnoreCase(s'):
controlla se s, s' sono uguali ignorando la differenza
maiuscole/minuscole. Qui s,s' sono gli oggetti legati rispettivamente
a this" e a altroAnimale. Il metodo equals() compare in tutte le classi

```

```

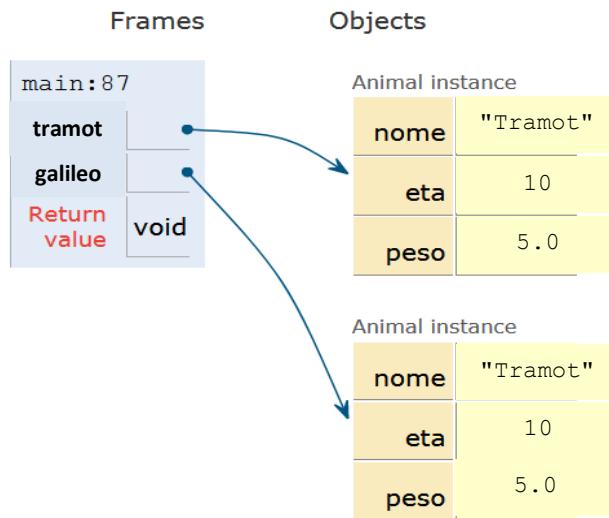
Java, e può essere ridefinito da noi. Quando definite una classe,
pensate sempre se è il caso di ridefinire un tale metodo.*/
public boolean equals(Animal altroAnimale){
    return
        (this.nome.equalsIgnoreCase(alteroAnimale.nome))
        &&
        (this.eta == alteroAnimale.eta)
        &&
        (this.peso == alteroAnimale.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 tramot = new Animal("Tramot",10,5.0); //valori
significativi
        Animal galileo = new Animal(); //valori di default

        System.out.println( "1. Tramot" );
        System.out.println(tramot);
        // sta per: System.out.println(tramot.toString());
        System.out.println( "2. Galileo" );
        System.out.println(galileo);
        /** All'inizio i due oggetti sono diversi */
        System.out.println("3. Tramot e' uguale a
Galileo?"+tramot.equals(galileo));
        /** Se assegno il primo al secondo attributo per attributo
diventano uguali attributo per attributo. */
        System.out.println("4. Assegno gli attributi di Tramot a Galileo
");
        galileo.assegna(tramot);
        System.out.println("5. Tramot e' uguale a
Galileo?"+tramot.equals(galileo));
        //Vero: stessi attributi
        System.out.println("6. Tramot == Galileo? "+(tramot==galileo));
        //Falso: diversi indirizzi
    }
}

```

Nel diagramma stack + heap qui sotto vedete la situazione della memoria alla fine dell'esecuzione del main di AnimalDemo. Dopo aver assegnato tutti gli attributi di "tramot" agli attributi di "galileo", i due oggetti hanno gli stessi attributi ma continuano a occupare **aree di memoria differenti**.



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 modifichiamo `z` non modifichiamo `x` e viceversa. Come esempio ricopiamo e eseguiamo il programma qui sotto, senza spiegare in dettaglio come scriverlo. Si tratta di una variante `SpecieNuova` della classe `Specie`, a cui aggiungiamo il metodo "cambia" per chiedere all'oggetto a cui mandiamo il metodo di modificare 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 pubblica */
    /** 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". Se un dato e' riservato, basta togliere il suo metodo
"get" e l'attributo non e' piu' accessibile dall'esterno della
classe. */
public String getNome() {return nome;}
public int getPopolazione() {return popolazione;}
public double getTassoCrescita() {return tassoCrescita;}

private static Scanner tastiera = new Scanner(System.in);

public void leggiInput(){
    System.out.println( " nome = " );
    nome = tastiera.nextLine();

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

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

/** Metodo di conversione specie --> stringa */
public String toString(){
    return " nome = " + nome + "\n popolazione = " + popolazione +
           "\n tasso crescita = " + tassoCrescita;
}

public int prediciPopolazione(int anni){
}

```

```

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

/** Questo metodo dinamico assegna gli attributi di this agli
attribuiti dell'oggetto "altraSpecie" passato come argomento.*/
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
direttamente == tra gli oggetti non va sempre bene, perche' ==
confronta gli indirizzi dei due oggetti, invece qui vogliamo
confrontare i valori dei attributi */
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 e eseguibile pubblica

    private static void pause() {
        /** Questo metodo ferma l'esecuzione del programma e aspetta un a
        capo per continuare. E' statico, quindi non viene inviato a un
        oggetto, ma chiamato scrivendo: pause(); */
        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" );
System.out.println(specieTerrestre);
//sta per: System.out.println(specieTerrestre.toString());
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");
System.out.println(specieKlingon);
pause();

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

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

System.out.println("\n 7. Vediamo ora un altro modo di modificare
gli oggetti");
System.out.println("Creo \"specieAfricana\" e le assegno i valori
Elefante");
SpecieNuova specieAfricana = new SpecieNuova(); //terzo oggetto
/** Copio i dati da specieTerrestre in specieAfricana */
specieTerrestre.cambia(specieAfricana);
System.out.println(specieAfricana);
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);  

System.out.println(specieAfricana);  
  

System.out.println("\n 12. NON modifico anche la specie Klingon  

perche' i due oggetti hanno indirizzi diversi: ");  

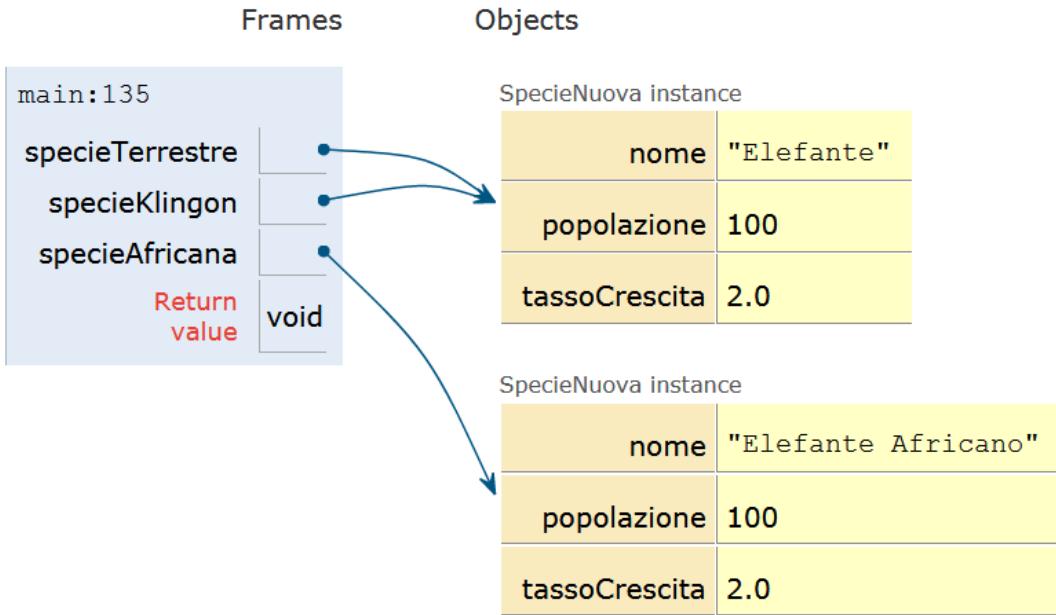
System.out.println(specieKlingon);  

}  

}

```

Nel diagramma stack + heap qui sotto vedete la situazione alla **fine dell'esecuzione** della classe **SpecieNuovaDemo**. Per effetto dei comandi di cui abbiamo parlato, i primi due oggetti "**SpecieTerrestre**" e "**SpecieKlingon**" si trovano nella stessa area di memoria, il terzo "**SpecieAfricana**" no, e le modifiche sul terzo non si ripercuotono sui primi due.



Soluzione dell'esercizio proposto nella Lezione 02 (30 minuti). Questa esercizio è piuttosto semplice, e la soluzione non sempre viene presentata a lezione; 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

    /** 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;

    public Rettangolo(double b, double h) //Costruttore di rettangoli
    {base = b; altezza = h; area = b*h;}

    private static Scanner tastiera = new Scanner(System.in); /* Questa
     * variabile e' statica, cioe' non appartiene a nessun rettangolo */
}
```

```

public void setDimensioni(double b, double h)
/** Per modificare base, altezza e area ora e' necessario un metodo
"set" (uno per attributo, o uno solo per modificare tutti) */
{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(){
    System.out.println( " base = " );
    base = tastiera.nextDouble();tastiera.nextLine();
    /* nextLine() consuma il carattere "return" */
    System.out.println( " altezza = " );
    altezza = tastiera.nextDouble();tastiera.nextLine();
    area = base*altezza;
}

/** Metodo di conversione rettangolo --> stringa */
public String toString()
{return " base      = " + base + "\n altezza = " + altezza +
"\n area      = " + area; }

/** La classe RettangoloDemo, che da' il nome al file */
public class RettangoloDemo {
public static void main(String[] args){
    Rettangolo R = new Rettangolo(2,2);
    //R nasce come rettangolo 2x
    System.out.println( "Inserisci nuovi valori per R" );
    R.leggiInput();
    System.out.println( "Valori inseriti di R \n" + R);
    //Se R viene concatenato con una stringa, viene interpretato come
    //R.toString()

    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

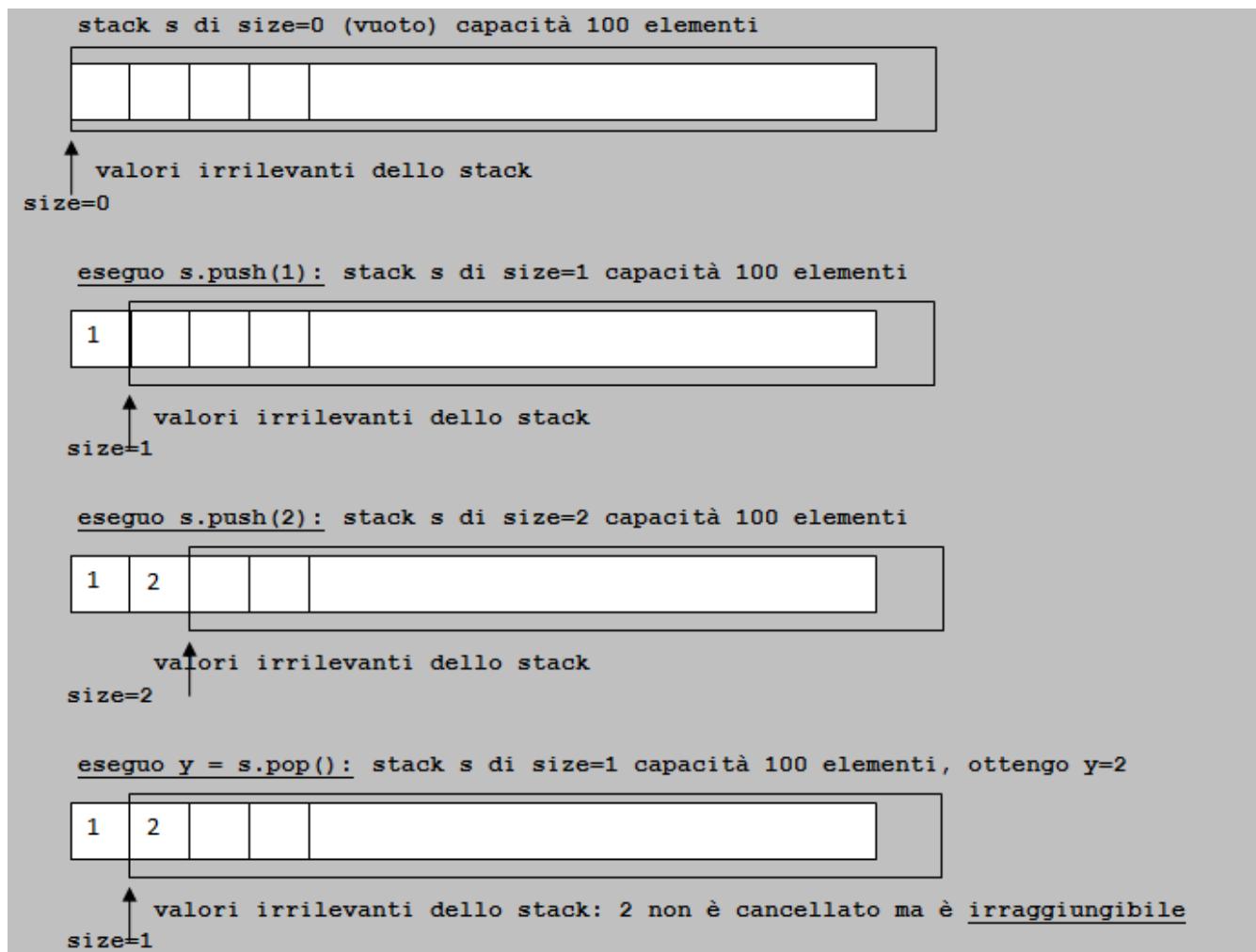
```

```
scrivere: */
R.setDimensioni(5,5);
System.out.println( "Valori modificati di R \n" + R);
}
```

Lezione 04

La classe "Stack", chiamate di metodi

Lezione 04. Parte 1. Un primo esempio di libreria: la classe Stack (50 minuti). Vi ricordiamo che uno **stack (o pila)** è una struttura dati in cui gli elementi vengono inseriti/rimossi secondo la politica **LIFO (Last-In-First-Out)**: l'ultimo elemento inserito è il primo a essere rimosso. Uno stack nasce vuoto e ha dimensione variabile, con la possibilità di aggiungere un nuovo valore x a fine pila (operazione **push(x)**) oppure di togliere un valore dalla pila e leggerlo (operazione **pop()**). Vediamo un esempio:



Nel disegno, quando aggiungiamo degli elementi 1,2 a uno stack per considerarli una informazione rilevante dobbiamo aumentare il valore size che indica parte rilevante dello stack: aggiungiamo 2 elementi e

portiamo il size a 2. Per "cancellare" l'ultimo elemento ci basta riportare il size a 1, l'elemento di posizione 1 diventa irrilevante e irraggiungibile.

Nella Lezione 02, abbiamo visto uno stack per rappresentare la memoria interna di una calcolatrice. Ora vediamo una classe Stack che provvede una libreria di operazioni per uno stack di interi, utilizzabile da altre classi. Nella nostra versione (semplificata) 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, né intralciare il corretto funzionamento di uno stack eseguendo operazioni non corrette. Questa tecnica di programmazione viene detta **incapsulamento dei dati** o **information hiding**.

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. L'**"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). Sta a voi controllare nel vostro strumento di Interactive Development Environment (quello con cui compilate e eseguite i programmi) 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 StackDemo.java

class Stack {
    private int[] stack; // inizialmente stack e' l'array vuoto (null),
    // 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;

    /* Notate che la falsita' della condizione capacity >=0 non dipende
    da un eventuale errore di programmazione (della libreria Stack stessa
    o del codice Client), ma da un eventuale dato errato inserito da chi
    usa il programma. Questa condizione, in particolare, dovrà essere
    gestita in modo che non fallire il programma a runtime se vengono
    inseriti dati sbagliati. A tal fine useremo il meccanismo delle
    'eccezioni', di cui parleremo piu' avanti. */

    // 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 stack, 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<stack.length) sarebbe scorretto! La dimensione
    // dello stack e' size, non l'intera lunghezza del vettore stack
    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);
    }
}

```

```

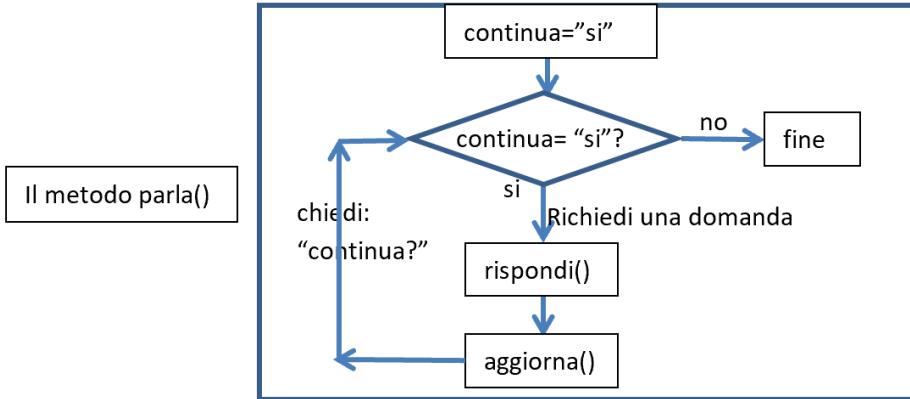
System.out.println(" s.full()      = " + s.full());
System.out.println(" s.empty()     = " + s.empty());
System.out.println(" s.equals(t) = " + s.equals(t));

Stack r = new Stack(3);
r.push(0); r.push(0); r.push(0);
Stack q = new Stack(3);
System.out.println("r non è vuoto!");
System.out.println(" r.full()      = " + r.full());
System.out.println(" r.empty()     = " + r.empty());
System.out.println(" r.equals(q) = " + r.equals(q));
/* LO STACK r contiene 3 elementi, tutti uguali a 0;
   LO STACK q non contiene elementi (il suo array 'stack'
   contiene 3 zeri ma non sono raggiungibili)
*/
}
}

```

Lezione 04. 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 "**maestri**", semplici robot virtuali che "dialogano" con l'utente (per finta, usando un trucco da prestigiatore).

Un maestro 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 Maestro 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. Nel prossimo disegno traduciamo questa descrizione in **un diagramma di flusso per il metodo *parla()*.**



(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: MaestroDemo.java
import java.util.Scanner;

class Maestro { //classe non eseguibile
    private String vecchiaRisposta = "la risposta è nel tuo cuore";
    private String nuovaRisposta;
    private String domanda;
    private static Scanner tastiera = new Scanner(System.in);

    public void parla(){
        String continua = "si";
        while (continua.charAt(0)=='s' || continua.charAt(0)=='S'){
            System.out.println( "Cosa vuoi chiedere? " );
            domanda = tastiera.nextLine();
            rispondi();
            // il metodo rispondi() è chiamato sull'oggetto this;
            // this = oggetto su cui viene chiamato il metodo
            // dinamico rispondi() (this lasciato implicito)
            /* Scrivere
               this.rispondi(); // this esplicito
               sarebbe corretto, perché equivalente alla versione precedente,
               solo con il this sintatticamente esplicito. */
            /* Invece scrivere: yoda.rispondi();
               sarebbe proprio SBAGLIATO, perché 'yoda' è un nome (una
  
```

```

variabile locale) del main, quindi non è visibile a questo
metodo parla(). Ricordiamo che this è una variabile
speciale di ogni metodo dinamico, che assume il valore
dell'oggetto su cui viene chiamato il metodo stesso nel
chiamante (nel nostro main, per esempio, this assume il valore
= l'indirizzo dell'oggetto puntato dalla variabile 'yoda'). */

aggiorna(); //oppure: this.aggiorna
// il metodo aggiorna() è chiamato sull'oggetto this
System.out.println( "Vuoi continuare?" );
continua = tastiera.nextLine();
}

System.out.println( "Il maestro ora riposa" );
}

/* I metodi che seguono sono considerati come metodi di servizio,
per cui dichiarati 'private': non visibili al codice client.*/
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 MaestroDemo{
// Classe eseguibile pubblica. Salvare in: MaestroDemo.java
public static void main(String[] args){
// Costruiamo un singolo maestro e iniziamo a parlare con lui
    Maestro yoda = new Maestro();
    yoda.parla();
}
}

```

Il trucco per ottenere un dialogo apparentemente sensato e' il seguente. La prima risposta è già scelta. Chi pone la domanda al maestro, invia la risposta alla prossima domanda facendo finta di suggerire la risposta alla domanda presente.

Fornendo i seguenti input:

```
Qual è il significato dell'esistenza  
Fai la cosa giusta  
sì  
Cosa devo fare nella vita  
Conosci te stesso  
sì  
C'e qualcosa che devo assolutamente conoscere?  
Non cercare una risposta a tutto  
no
```

Si ottengono i seguenti input/output:

Cosa vuoi chiedere?

Qual è il significato dell'esistenza

Avrei bisogno di un suggerimento: cosa mi suggerisci?

Fai la cosa giusta

Hai posto la domanda : Qual è il significato dell'esistenza

Ecco la tua risposta : la risposta è nel tuo cuore

Vuoi continuare?

sì

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?

sì

Cosa vuoi chiedere?

C'e qualcosa che devo assolutamente conoscere?

Avrei bisogno di un suggerimento: cosa mi suggerisci?

Non cercare una risposta a tutto

Hai posto la domanda : C'e qualcosa che devo assolutamente conoscere?

Ecco la tua risposta : Conosci te stesso

Vuoi continuare?

no

Il maestro ora riposa

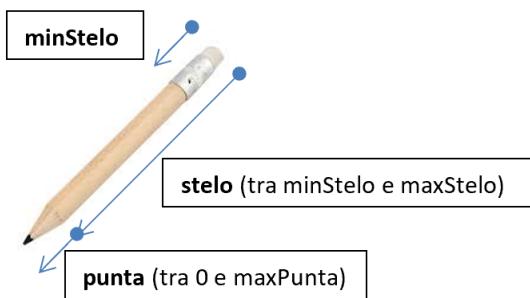
Esercitazione 01

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 interna in millimetri, da un minimo **`minStelo`** a un massimo **`maxStelo`**) seguita da una punta (un interno 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`**. Il prossimo disegno riassume il significato appena visto dei diversi attributi.



(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.

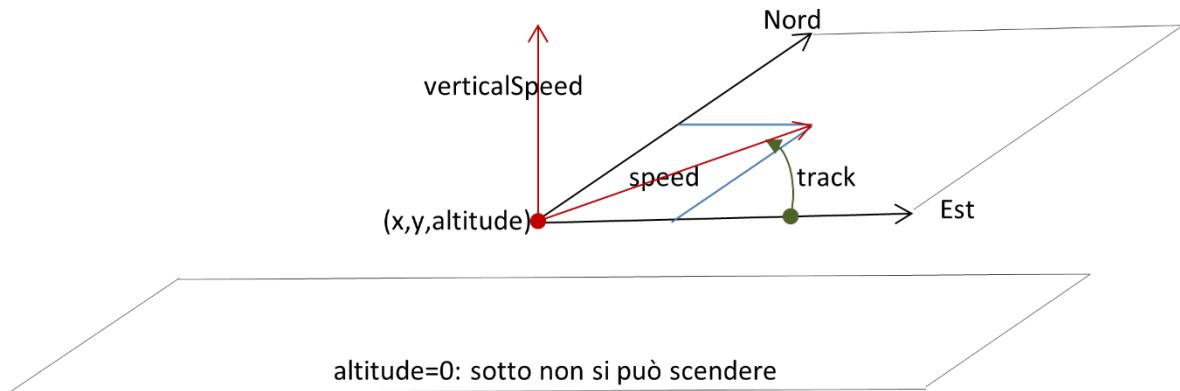
Scrivete **Matita.maxStelo** per richiamare il massimo dello stelo (attributo statico). Scrivete **Math.min** per richiamare il metodo statico **min(x,y)** della classe Math, che calcola il minimo. Includiamo una classe TestMatita per sperimentare la classe Matita: eseguitela e controllate che i risultati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

```
//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 (interi, in km): **x,y** e **altitude** (non negativa), due velocità (interi, 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). Il prossimo disegno riassume i significati degli attributi di un elicottero.



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.

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 Hiding

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**. Il valore di pi-greco è un attributo costante della classe Math e si richiama scrivendo **Math.PI**.

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

    /* 0 <= capacita AND 0 <= livello <= capacita
    rappresenta un INVARIANTE di classe, ovvero una proprietà che definisce
    la buona formazione degli oggetti della classe Bottiglia: questa
    proprietà deve essere mantenuta vera dopo l'esecuzione del costruttore
    e dei metodi che manipolano gli oggetti di tipo Bottiglia. */

    public Bottiglia(int capacita){
        this.capacita = capacita;
        /** "this.capacita" e' un attributo di bottiglia mentre "capacita"
        e' l'unico argomento del costruttore Bottiglia */
        livello = 0;
        assert (0<=livello) && (livello <= capacita);
        // per assicurare che l'oggetto sia ben formato
    }

    /* Tutti i metodi che seguono sono dinamici (non-static), quindi hanno
    tutti un parametro 'this', oltre ai parametri espliciti elencati entro
    le loro parentesi (...). Il 'this' prende come valore l'indirizzo
    dell'oggetto su cui viene chiamato il metodo. Ricordiamo che la parola-
    chiave this può essere lasciata implicita ovunque non si creino
    ambiguità.

    Un esercizio che si può fare è individuare in quali punti c'è il this
    implicito (e dove quindi si potrebbe rendere esplicito). */

    /** 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);
    // dopo l'esecuzione del metodo l'oggetto deve essere ancora
    // ben formato <-- RISPETTIAMO L'INvariante di CLASSE STABILITO
    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);
    // dopo l'esecuzione del metodo l'oggetto deve essere ancora
    // ben formato <-- RISPETTIAMO L'INvariante di CLASSE STABILITO
    return rimossa;
}

public int getCapacita(){ return this.capacita; }
public int getLivello() { return this.livello; }

// Non consentiamo di cambiare la capacita', quindi non c'è
// un metodo setCapacita()

public void setLivello(int livello){
    this.livello = livello;
    /** "this.livello" e' un attributo di bottiglia mentre "livello" e'
    l'unico argomento del metodo setLivello */
    assert (0<=livello) && (livello <= capacita);
}

```

```

}

public String toString() //conversione oggetto bottiglia --> stringa
{return livello + "/" + capacita;}
}

```

Rappresentiamo le operazioni lecite dell'indovinello **The Water Jug Riddle** in modo object-oriented con tre metodi statici: "**riempি, svuota, travasa**" di argomenti di tipo Bottiglia. La definizione 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 sono dichiarati statici. */

public static void riempি(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
    // non di piu' perche' non vada persa acqua
    int capienzaResiduaB = b.getCapacita() - b.getLivello();
    /* rimuovo questa quantita da a, o tutto da a se a non basta a
    riempire b. Aggiungo la quantita' ottenuta alla bottiglia b */
    b.aggiungi(a.rimuovi(capienzaResiduaB));
    //una soluzione alternativa: a.rimuovi(b.aggiungi(a.getLivello()));
}

public static void descrizione(String m, Bottiglia b3, Bottiglia b5)
{System.out.println(m + "\n" + b3 + "\n" + b5);}

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);
}
}

```



Un'immagine del "Water Jug Riddle"

Lezione 05. Parte 2. Un esempio di encapsulamento di dati: la classe Frazione. (40 minuti). Riprendiamo l'argomento di encapsulamento dei dati, e definiamo una classe **Frazione** per rappresentare le frazioni intere e le operazioni su di esse. I calcoli su frazioni tendono a produrre numeratori e denominatori **molto grandi**: per evitarlo, nella nostra classe una frazione viene sempre **ridotta ai minimi termini**. Per semplicità di rappresentazione, useremo solo frazioni **con denominatore 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 molte 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*/
    /** Metodo statico MCD per il calcolo del Massimo Comun Divisore
    per interi a,b non entrambi nulli.
```

Chi ha progettato la classe Frazione ha deciso di definire MCD come metodo **statico** perché MCD è un metodo che viene applicato ***direttamente*** a due int e solo ***indirettamente*** a un oggetto di tipo Frazione. Per esempio, il metodo dinamico **semplifica()**, definito sotto, chiama MCD e lo applica al numeratore e denominatore di una frazione, visti come valori di tipo int. */

```
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: semplifica e rende positivo il denominatore */
public void setFrazione(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 di conversion frazione --> stringa */
public String toString(){
    if (den != 1)
        return num + "/" + den;
    else /* den=1 */
        return num + "";
    /* Al posto di (num/1) scrivo num, trasformato in stringa grazie a
una concatenazione con "" */
}

/** 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);
}

/** Il prossimo metodo sommaS è una versione statica del metodo di
somma tra due frazioni. SommaS usa lo stesso algoritmo di Somma, ma
SommaS non ha il this e un parametro esplicito f di tipo Frazione,
bensì due parametri esplicativi a,b di tipo Frazione. Cambia quindi la

```

modalità di chiamata: per un esempio, si vedano i test inclusi nella classe FrazioneDemo.java. Nella classe Frazione preferiamo però la versione "dinamica" Somma, perché privilegiamo la progettazione object-oriented. Torneremo su questi argomenti più volte nel corso. */

```

public static Frazione sommaS(Frazione a, Frazione b){
    int n= a.num*b.den + a.den * b.num;
    int d= a.den*b.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      (classe per 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( "\n t,u,v valgono" );
        System.out.println(t + "\n" + u + "\n" + v);

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

        //Chiamata della versione statica sommaS della somma
        //della somma:
        Frazione h = Frazione.sommaS(Frazione.sommaS(t,u), v);
        System.out.println(h);

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

```
// Controllo che a=3/4 e b=30/40 siano uguali
System.out.println( "\n Controllo che 3/4 = 30/40");
Frazione a = new Frazione(3,4), b = new Frazione(30,40);
System.out.println( "a=new Frazione(3,4)    vale: " + a);
System.out.println( "b=new Frazione(30,40) vale: " + b);
System.out.println( "a.equals(b)="+a.equals(b));

System.out.println
( "\n Invece a= " + a + " e v=" + v + " sono diversi");
System.out.println( "a.equals(v)="+a.equals(v));

}
```

Lezione 06

Classi di array di oggetti

Lezione 06 (*90 minuti*). In questa lezione definiamo Rubrica, il primo esempio di classe di **array di oggetti** (vedi Cap.9.6 del Savich). Definiamo un contatto come la coppia di un nome e di un indirizzo email, e una rubrica come un array di contatti. La classe Rubrica avrà un **invariante** ("una rubrica è un vettore parzialmente riempito privo di contatti ripetuti"), il cui mantenimento è **essenziale per il corretto funzionamento della classe**.

La classe Contatto ha attributi privati, metodi get e set senza restrizioni e un metodo di scrittura. **Nota**. Visto che non poniamo restrizioni all'accesso agli attributi della classe Contatto, potremmo dichiararli pubblici. Tuttavia, per maggiore uniformità, si preferisce anche in questo caso dichiararli privati.

```
// Contatto.java (costruttore a 2 argomenti, metodi get, set, 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;}

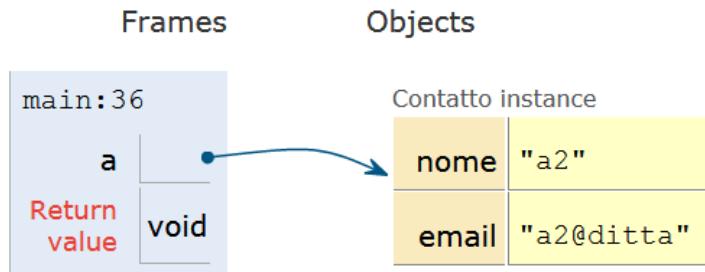
    // Trasforma un contatto nella stringa che lo descrive
    public String toString()
    {return " - " + nome + " : " + email;}
}

// 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);
```

```

System.out.println( "Modifico nome a in a2" );
a.setName( "a2" );
System.out.println( "Contatto a modificato: " + a );
System.out.println( "Modifico email a in a2@ditta" );
a.setEmail( "a2@ditta" );
System.out.println( "Contatto a modificato: " + a );
}
}

```



Il contatto di nome "a" e di email "a@ditta", dopo che abbiamo modificato nome ed e-mail. Il contatto a si trova sullo stack ed è l'indirizzo dell'area della heap che contiene nell'ordine: il suo nome e email

La classe **Rubrica** ha come attributi privati: **(i)** un array "contatti" di contatti parzialmente riempito, nei posti da 0 a numContatti-1, e **(ii)** un intero "numeroContatti" che ci dice quanti contatti sono stati inseriti nel array.

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 array 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 oggetto costruito e che viene preservata da ogni applicazione di ogni metodo. Definiamo Rubrica in modo da rendere vero il seguente invariante: *i posti occupati vanno da 0 a numContatti-1, ogni nome compare in al più un contatto di una rubrica, e (0 ≤ numContatti ≤ lunghezza array 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. Questo vuol anche dire, però, che se **per errore non rispettiamo questa parte dell'invariante** a un nome corrisponderanno più email, e la classe **non funzionerà** in

modo corretto. In questo caso potrei cercare l'email di un nome, ottenere una risposta, ma quella risposta può non essere l'unica risposta possibile, e potrei aver mancato proprio la risposta che mi interessa.

Rubrica ha un metodo privato int cercaIndice(String n) per ottenere l'unica posizione di un nome nell'array contatti: è un intero tra 0 e numContatti-1. Restituisce numContatti se il nome non compare, e identifica i nomi a meno di maiuscole/minuscole. Il metodo è privato, viene usato per definire i metodi pubblici della classe.

Rubrica ha metodi pubblici:

- (i) **String toString()** per generare una stringa che descrive 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 rubrica ha posizioni
occupate da 0 a numContatti-1 e in queste posizioni non contiene lo
stesso nome (a meno di maiuscole/minuscole) due volte, (ii)
(0<=numContatti <= lunghezza array contatti) */

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

public Rubrica(int maxContatti){
    assert (maxContatti >=0):
    "errore maxContatti negativo: " + maxContatti;
//inizializza 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 sono null
/** La nuova rubrica costruita soddisfa l'invariante. */
}

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

public String toString(){ // conversione Rubrica-->Stringa
    int i=0;
    String s = "Num. contatti = " + numContatti + "\n ";
    // concateniamo i contatti di indice da 0 fino a numContatti-1.
    /* Gli altri contatti sono privi di significato,
    non esistono dal punto di vista logico */
    while(i<numContatti){s = s + contatti[i].toString();++i;}
    // si noti che contatti[i].toString() chiama la toString()
    // della classe Contatto
    return s;
}

/** 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, usando l'indice restituito da cercaIndice(n),
con il rischio di contraddirre 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;}
    // L'oggetto su cui è chiamato il metodo equalsIgnoreCase()
    // è di tipo String, ed è stato restituito dalla chiamata
    // contatti[i].getNome(). Questa chiamata applica il metodo getNome()
    // su contatti[i], a sua volta un oggetto di tipo Contatto.

    // 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 nell'array
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

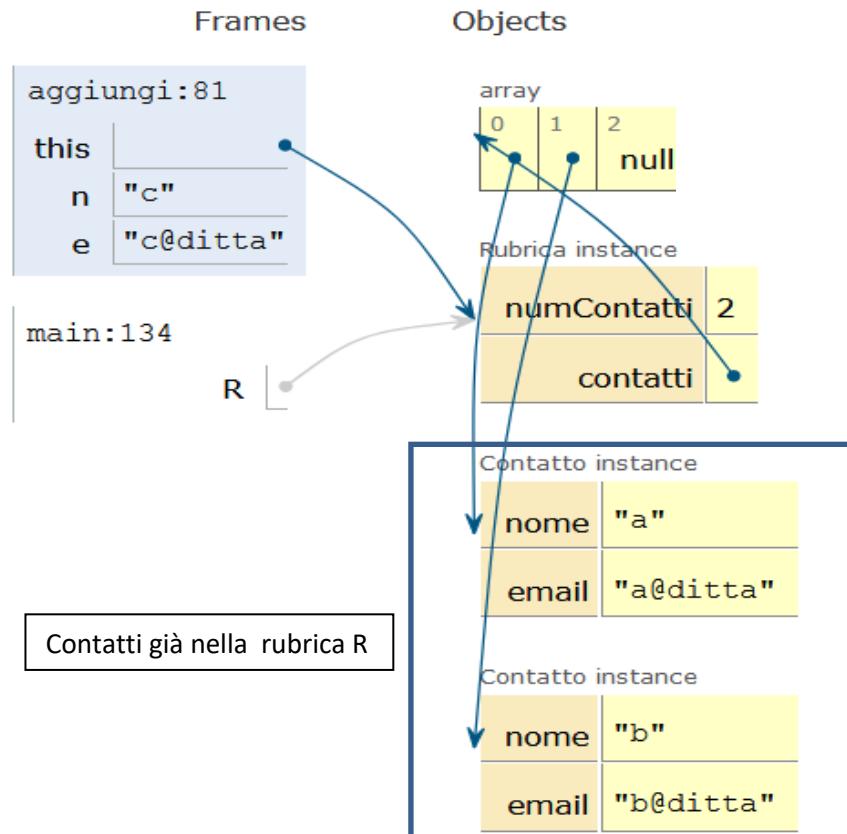
    /** Se invece il contatto c'e': diminuiamo di 1 il numero dei
    contatti e spostiamo il contatto al fondo (ora irraggiungibile) al
    posto del contatto i da cancellare. Se i e' l'ultimo contatto allora
    i resta al fondo e resta non piu' raggiungibile. */
    --numContatti;
}

```

```
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 contatto 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;
}
}
```



L'effetto del metodo `aggiungi()` sulla memoria. Il metodo "aggiungi" aggiunge il contatto di nome "c" e email "c@ditta" alla rubrica R che contiene i contatti a e b. Nella parte dello stack relativa al metodo, `this=R` punta a un'area di memoria che include la dimensione della rubrica, e l'indirizzo del vettore che include gli indirizzi attuali dei contatti di nome "a" e "b". A loro volta i contatti sono coppie di informazioni nome/email.

```
//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");
        System.out.println(R);
        //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" ); System.out.println(R);
        System.out.println( "(3) Aggiungo b (ma c'e' gia'): successo = "
+ R.aggiungi( "b", "e" )); System.out.println(R);
        System.out.println( "(4) Modifico b in b2: successo = "
+ R.modifica( "b", "b2" )); System.out.println(R);
    }
}
```

```

+ R.cambiaNome( "b" , "b2")); System.out.println(R);
System.out.println( "(5) Modifico b@ditta in b2@ditta: successo =
" + R.cambiaEmail( "b2" , "b2@ditta")); System.out.println(R);
}
}

```

Fine Lezione 06

Nota su: rimozione di un contatto da una rubrica e alias
Attenzione, questa discussione 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 assegniamo 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.

In un corso precedente, uno studente ha chiesto se il metodo **boolean rimuovi(String n)** visto in questa lezione, che rimuove un contatto da una rubrica, crea indirizzi duplicati per lo stesso contatto, ovvero degli "alias". *Rispondo ma solo per curiosità, la risposta non è essenziale per un corso del primo anno.* Ricopiamo qui il metodo:

```

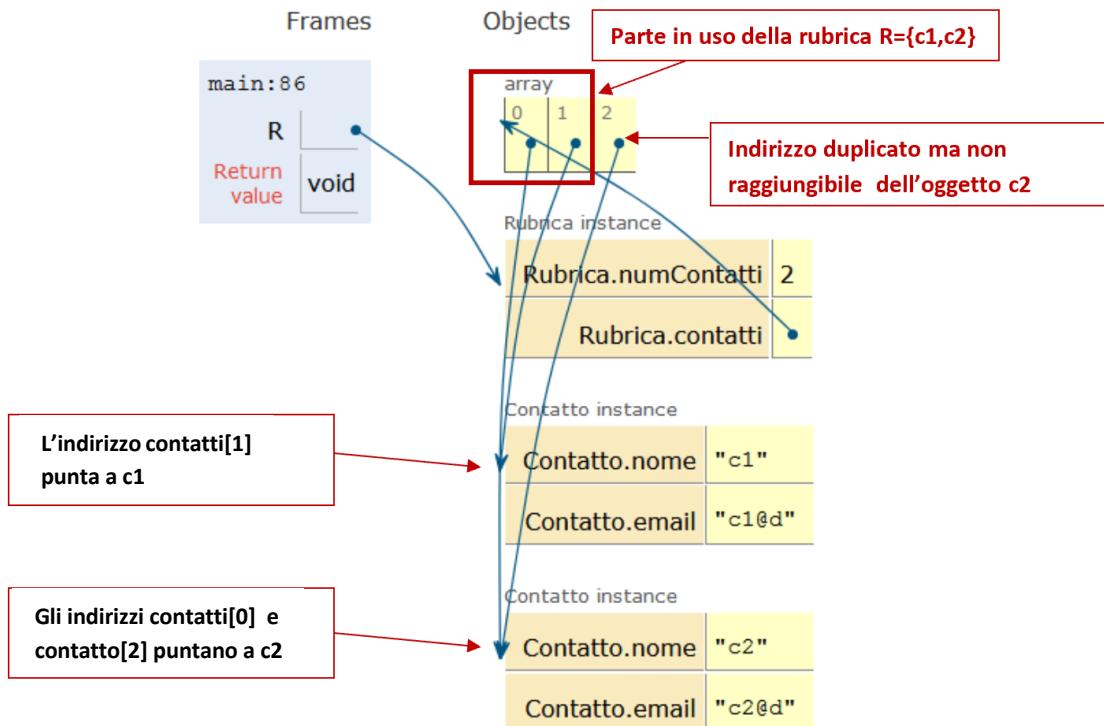
public boolean rimuovi(String n){
    int i = cercaIndice(n);
    if (i == numContatti) return false;
    --numContatti;
    contatti[i]=contatti[numContatti];
    return true;
}

```

La risposta è: il metodo **rimuovi crea degli alias, ma non crea degli alias effettivamente raggiungibili**. 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) c0, c1, c2. In questo caso numContatti=3. Supponiamo di cancellare il contatto c0. Allora assegniamo numContatti=2. Inoltre assegniamo contatto[0] = contatto[2], dunque adesso c0 è sovrascritto, contatto[0] vale c2, e contatto[1] vale c1, e contatto[2] vale c2. Abbiamo creato un alias, dato che contatto[0] e contatto[2] contengono lo stesso indirizzo c2. **Tuttavia l'alias non**

è raggiungibile dai metodi della classe. La variabile contatto[2] è ora irraggiungibile dalla classe Rubrica, perché si trova nella posizione 2, e dato che numContatti=2, la posizione 2 non fa parte delle posizioni utilizzabili dai metodi della Rubrica.

Possiamo riassumere tutto quanto abbiamo appena detto con un disegno Frames+Objects:



È possibile che in futuro io possa raggiungere l'indirizzo duplicato dell'oggetto `c2`, quello che si trova in `contatto[2]`? No, non posso. Potrei 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 `c2` in `contatto[2]` viene sovrascritto. In nessun caso il secondo indirizzo `c2` contenuto in `contatto[2]` può venir utilizzato da un programma che usa classe `Rubrica`, quindi questo secondo indirizzo **non costituisce un alias che possiamo effettivamente usare.**

Lezione 07

Diagrammi UML e array estendibili

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

Scrivere programmi è un lavoro di gruppo 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 è grafico e si chiama **UML** (**Unified Modeling Language**). Non preoccupatevi, non vi chiediamo di saper scrivere diagrammi UML all'esame, ma talvolta useremo diagrammi UML a lezione per descrivere delle classi.

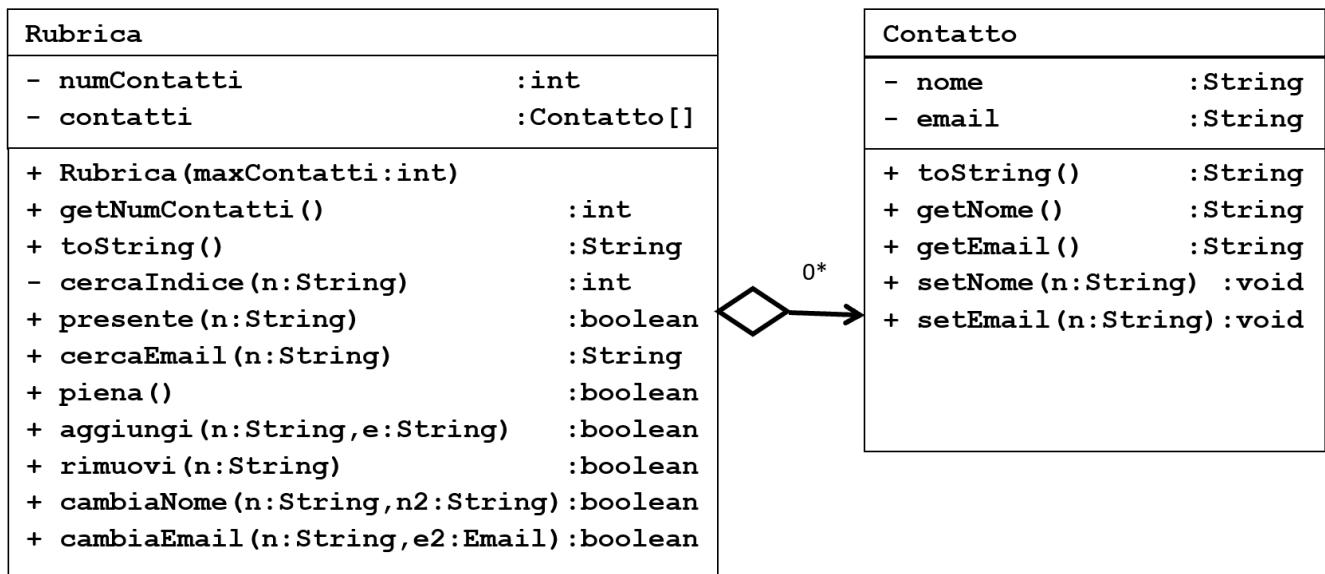
Classi in UML. Una classe viene descritta in UML con un rettangolo con tre sezioni orizzontali, il nome della classe, i suoi attributi seguiti da ":" e il loro tipo, e i metodi, seguiti dalla lista dei loro argomenti e da ":" e il loro tipo. I metodi pubblici hanno un +, i metodi privati un -, e attributi e metodi statici sono sottolineati. Come esempio prendiamo la classe Matita vista a Esercitazioni.

Matita
+ <u>minStelo</u> :int
+ <u>maxStelo</u> :int
+ <u>maxPunta</u> :int
- <u>stelo</u> :int
- <u>punta</u> :int
+ <u>Matita(stelo:int)</u>
+ <u>disegna()</u> :void
+ <u>tempera()</u> :void
+ <u>getStelo()</u> :int
+ <u>getPunta()</u> :int

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 degli 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). Come esempio, nella lezione precedente abbiamo visto il caso di **C = Rubrica e D = Contatto**.



Nel caso la classe D compaia solo come parte della classe C (per esempio, se utilizziamo D = Contatto solo come parte di C = Rubrica), allora diciamo che D è parte della **composizione** di C e indichiamo la relazione (più stretta dell'aggregazione) 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 possiamo aggiungere sopra la riga il nome e la direzione della relazione (con una freccia) tra C e D.

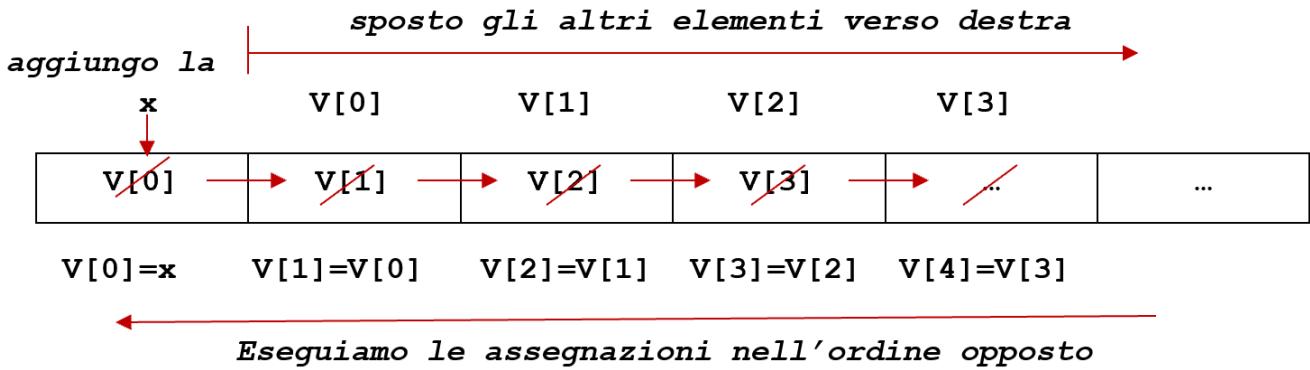
Lezione 07. Parte 2. Array estendibili (70 minuti). Definiamo una classe di array parzialmente riempiti, con la possibilità di aggiungere e togliere elementi. L'array raddoppia di dimensioni quando viene chiesto di aggiungere un elemento a un array già pieno. In questo caso, gli elementi già esistenti vengono ricopiatati nel nuovo array, l'indirizzo del vecchio array viene sovrascritto con l'indirizzo del nuovo array, e il vecchio array diventa irraggiungibile e verrà riciclato.

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

Diagramma UML di ArrayExt

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

Spostamento degli elementi di un array. Per spostare un segmento di elementi di array 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 $v[0], v[1], x, v[2], v[3]$. Ecco un disegno dello spostamento sinistra->destra:

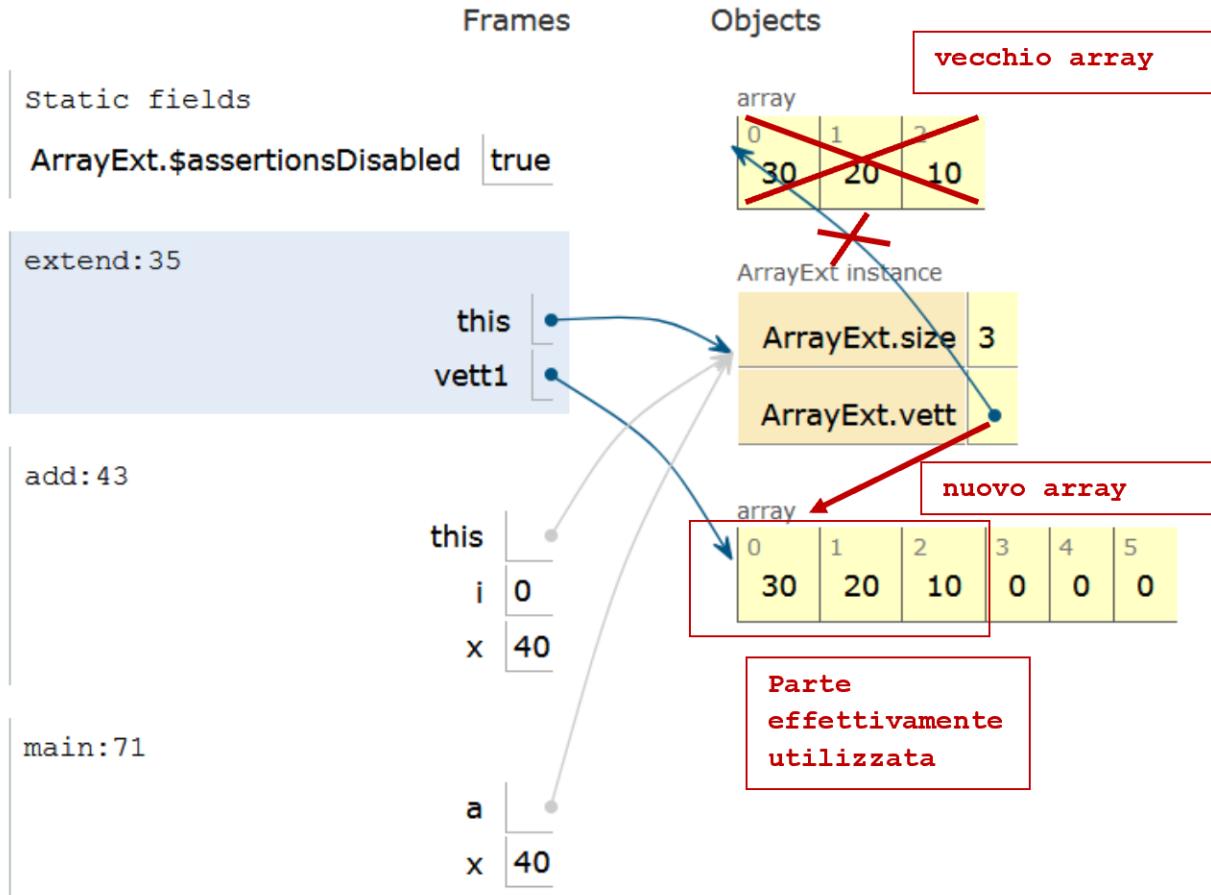


Nota. Se invece assegnassimo $v[1]=x, v[2]=v[1], v[3]=v[2], v[4]=v[3]$, muovendoci da sinistra a destra, otterremo: x, x, x, x, x . Non faremmo altro che ricopiare x più volte: non fate così!

Per la stessa ragione, se vogliamo realizzare lo spostamento degli elementi dell'array di un passo in direzione destra->sinistra, allora dobbiamo assegnare $v[j]=v[j+1]$ ma **muovendo j nella direzione opposta, sinistra->destra**. Se abbiamo $v[0], v[1], v[2], v[3]$ e vogliamo rimuovere $v[0]$, dobbiamo assegnare: $v[0]=v[1], v[1]=v[2], v[2]=v[3]$, in modo da ottenere $v[1], v[2], v[3]$. Eseguiamo le assegnazioni in ordine opposto. Ecco un disegno dello spostamento destra->sinistra:



Aggiungere un elemento 40 a un array `vett={10,20,30}` pieno ha come primo effetto di ***duplicare l'array e ricopiare gli elementi***. La vecchia copia viene abbandonata e il suo spazio di memoria riciclato, come mostrato nel prossimo disegno:



Con questa osservazione abbiamo terminato la descrizione della classe `ArrayExt`: ora la realizziamo.

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

Inoltre `ArrayExt` fornisce operazioni più generali, di aggiunta e rimozione di un elemento dato il suo indice. Queste operazioni mantengono l'ordine all'interno dell'array se questo esiste. Vengono utilizzate nel caso di array ordinati, quando vogliamo modificare l'array, ma facendo sì che resti ordinato. */

```

public class ArrayExt{
    // Invariante: (0 <= size <= lunghezza vett) e (lunghezza vett>0)
    // (parte significativa di vett: da 0 a size-1)
    private int size;
    //indica la parte effettivamente in uso del array, 0 all'inizio
    private int[] vett;
    public int getSize(){return size;}

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

    public String toString(){
        String s = " size = " + size;
        for(int i=0;i<size;++i) s=s+"\n vett["+i+"]="+vett[i];
        return s;
    }

    //Metodo di lettura dell'elemento i con 0<=i<size
    public int get(int i){
        assert 0<=i && i<size: "get su indice non in 0,...,size-1 " + i;
        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: "set su indice non in 0,...,size-1 " + i;
        vett[i]=x;
    }

    //Metodo per espandere l'array quando necessario
    private void extend(){
        int[] vett1 = new int[vett.length*2];
        //nuovo array di lunghezza doppia
        for(int i=0;i<size;++i)

```

```

        {vett1[i]=vett[i];} //trascrivo il vecchio array nel nuovo
        vett=vett1; //aggiorno l'indirizzo dell'array "ufficiale"
        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: "add su indice non in 0,...,size " + i;
    if (size==vett.length) //se manca lo spazio
        extend(); //espando l'array
    assert size<vett.length; //controllo che ora lo spazio ci sia
    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; //controllo l'invariante
}

//Rimozione della posizione 0<=i<size effettivamente nell'array.
//restituisce l'elemento rimosso e quindi puo' fare da "pop"
public int remove(int i){
    assert 0<=i && i<size : "set su indice non in 0,...,size-1 " + i;
    --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
    assert 0<=size && size<=vett.length; //controllo l'invariante
    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.
        String msg = "Aggiungo i valori x=0,1,...,11 al vettore a" +

```

```

"\n - sempre in prima posizione"      +
"\n - ognuno davanti ai valori precedenti" + 
"\n - ogni aggiunta sposta avanti di uno gli elementi precedenti";
System.out.println(msg);
int x=0;
while (x<12) {a.add(0,x);++x;}
System.out.println(" a \n" + a);
System.out.println
("Rimuovo a[0]=11 e sposto indietro di uno gli altri elementi");
System.out.println( " a.remove(0)=" + a.remove(0));
System.out.println(" a \n" + a);
System.out.println( "Aggiungo x=-1 in posizione 11 (in fondo)");
a.add(11,-1);
System.out.println(" a \n" + a);
}
}

```

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 array in un nuovo array 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)
```

Il metodo copyOf prende un array 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);}
```

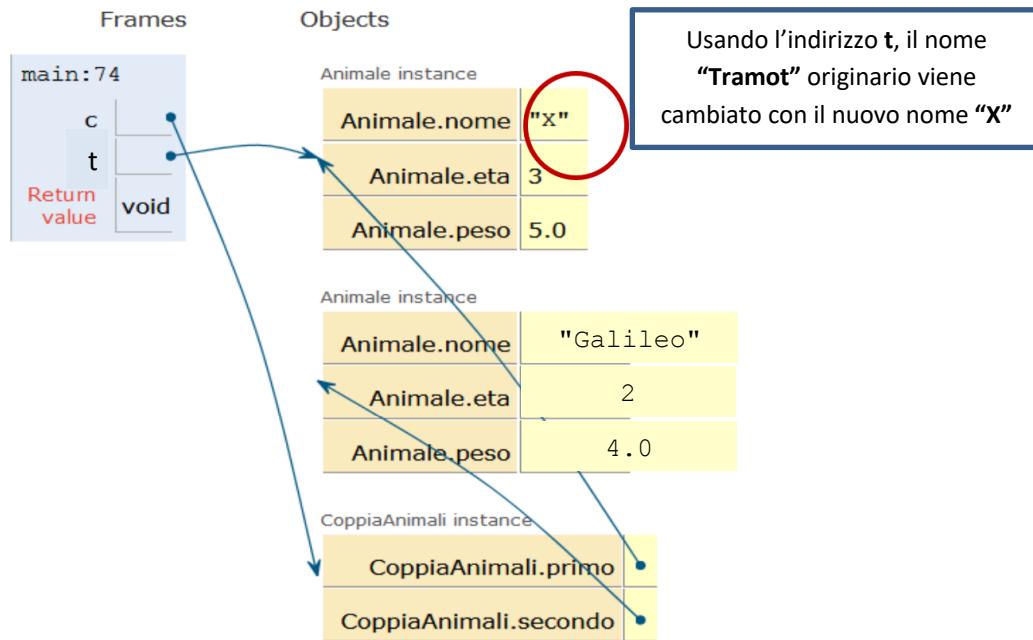
Lezione 08

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 si riesce a trovare un altro percorso al dato tramite i puntatori contenuti nella memoria dinamica (ovvero tramite degli alias) si può 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 ottenere una coppia di animali Tramot e Galileo, e cambiare il nome di "Tramot" **con un nuovo nome "X"**, aggirando il divieto di scrittura posto in CoppieAnimali. Facciamo come segue. Il metodo get di CoppieAnimali non consente di modificare l'indirizzo dell'oggetto Tramot originario, però fornisce una copia **t** di questo indirizzo. Basta passare questa copia f a un metodo set (di scrittura) della classe Animali per raggiungere in scrittura i dati di Tramot.

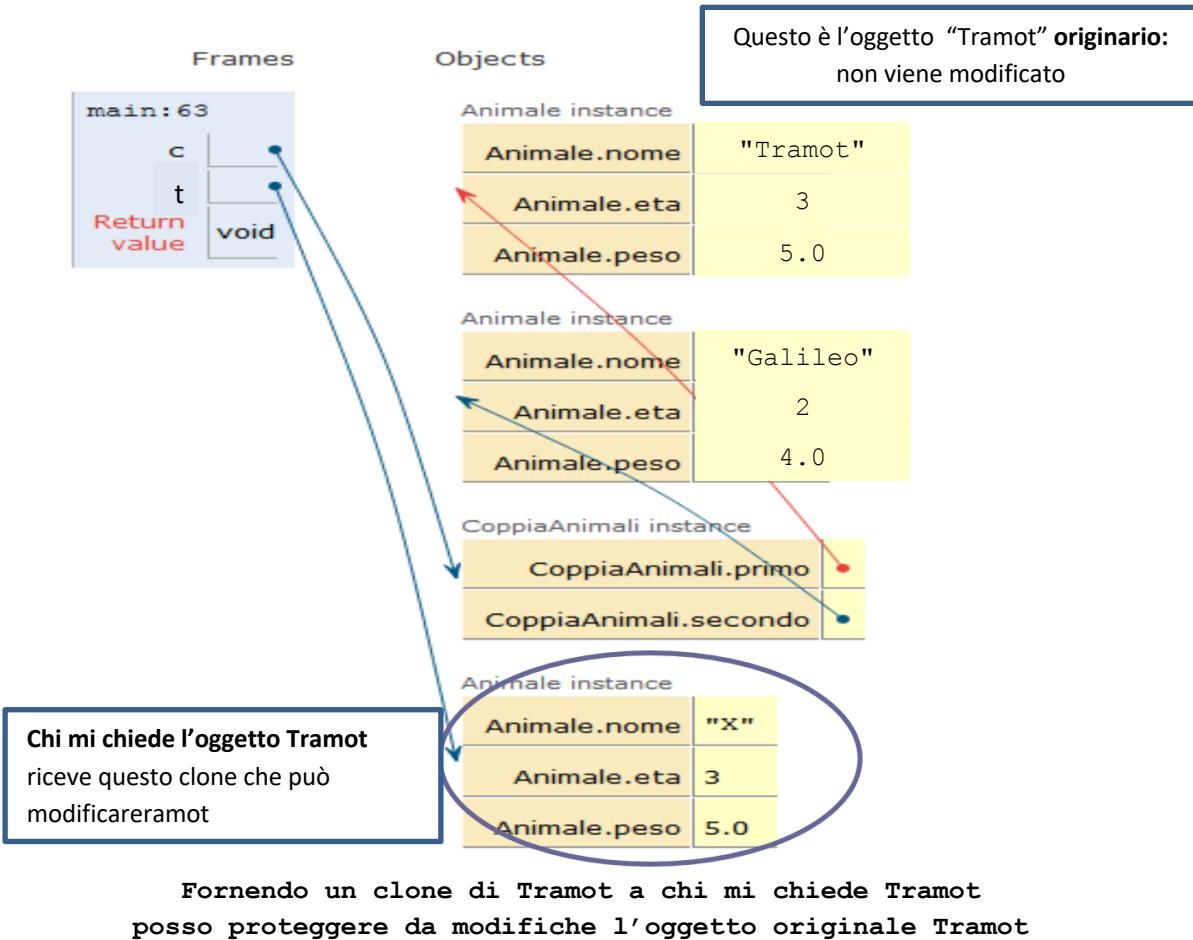
Il prossimo disegno riassume come è avvenuto il Security Leak: un diverso percorso attraverso i dati ha consentito di aggirare i divieti di scrittura.



Ottenendo una copia **t** dell'indirizzo di Tramot posso modificare Tramot scavalcando i divieti di scrittura posti dalla classe **CoppiaAnimali**.

Un modo sicuro per non consentire di modificare gli attributi dei due oggetti di tipo Animale dall'esterno è passare a richiesta l'indirizzo di una copia o "clone" di un oggetto-animale, con tutti gli attributi uguali all'oggetto originario, ma non l'indirizzo dell'oggetto originario.

Nel prossimo disegno vediamo la formazione del clone per proteggere dalla possibilità di modifica i dati originari.



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 String    toString()
}
```

```

{return " |nome=" + nome + "\n eta=" + eta + "\n peso=" + peso;}}
```

//CoppiaAnimali.java

```

public class CoppiaAnimali{
    private Animale primo;
    private Animale secondo;

    /** Un oggetto CoppiaAnimali dopo la sua dichiarazione ha come valori
    nei campi primo=null, secondo=null. All'inizio primo, secondo non sono
quindi valori corretti di tipo Animale */

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

//Hacker.java (Vediamo come aggirare i divieti di scrittura)

```

public class Hacker {
    public static void main(String[] args){
        System.out.println("\n Definisco la coppia c = Tramot e Galileo");
        CoppiaAnimali c =
            new CoppiaAnimali("Tramot",3,5.0,"Galileo",2,4.0);
```

```

System.out.println(c);
System.out.println("\n Chiedo una copia dell'indirizzo di Tramot");
Animale t = c.getPrimo();
System.out.println("Se getPrimo mi da' una copia dell'indirizzo
posso modificare Tramot");
System.out.println("Invece non si puo' modificare Tramot se
getPrimo mi da' un clone");
t.setNome("X");
System.out.println( "\n Ora la coppia c vale \n" + c);
System.out.println( "t viene modificato in ogni caso" );
System.out.println(t);
}
}

```

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. L'indirizzo indefinito, "null", si può vedere come un nodo speciale che non contiene/punta a nulla. Ogni altro nodo punta a un nodo definito in precedenza. Si tratta quindi di una definizione *ricorsiva*: si intende che i nodi sono la più piccola classe che si può ottenere in questo modo, contengono null, ogni nodo che punta a null, ogni nodo che punta a un nodo che punta a null, eccetera.

I nodi sono alla base di molte classi in programmazione dinamica: usando i nodi costruiremo per esempio le pile dinamiche (senza una dimensione massima). Ecco 3 nodi diversi da null, con elementi 7,5,3, ciascuno contenente oltre al un elemento l'indirizzo del nodo precedentemente creato. Il primo nodo creato contiene l'indirizzo null nel campo next:



La classe Node ha come attributi privati un elemento di tipo int e un oggetto next di tipo Node (che essendo un oggetto è un puntatore), 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;}
}

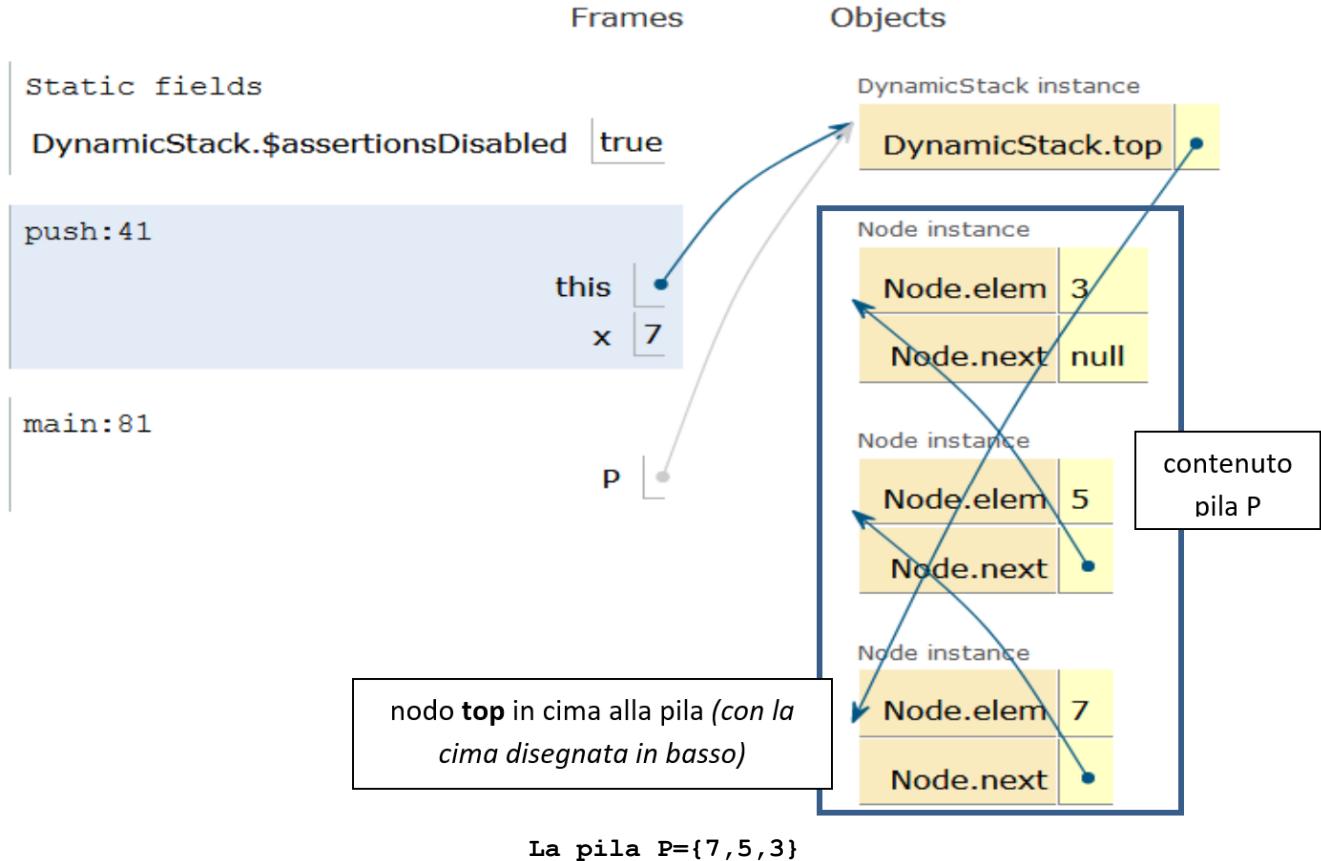
```

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 (in precedenza, avevamo visto le pile statiche, con una dimensione massima fissata all'inizio).

Una *pila dinamica* è una lista di nodi: n_0, n_1, \dots, n_{k-1} . Ogni nodo che contiene un elemento e l'indirizzo del nodo successivo tranne n_{k-1} che contiene l'indirizzo null. L'oggetto pila è descritto da un unico attributo top di tipo Node, che punta al primo nodo n_0 se $k>0$, oppure top=null se $k=0$ (se la pila è vuota). **In linea di principio, sarebbe possibile codificare** le pile **direttamente con i nodi**. La pila vuota sarebbe rappresentata da "null", l'elemento indefinito di Java, ma **inviare un metodo** pubblico **a null produce un errore**, quindi una tale rappresentazione non sarebbe conveniente, perché occorrerebbe gestire il caso della pila vuota come caso particolare.

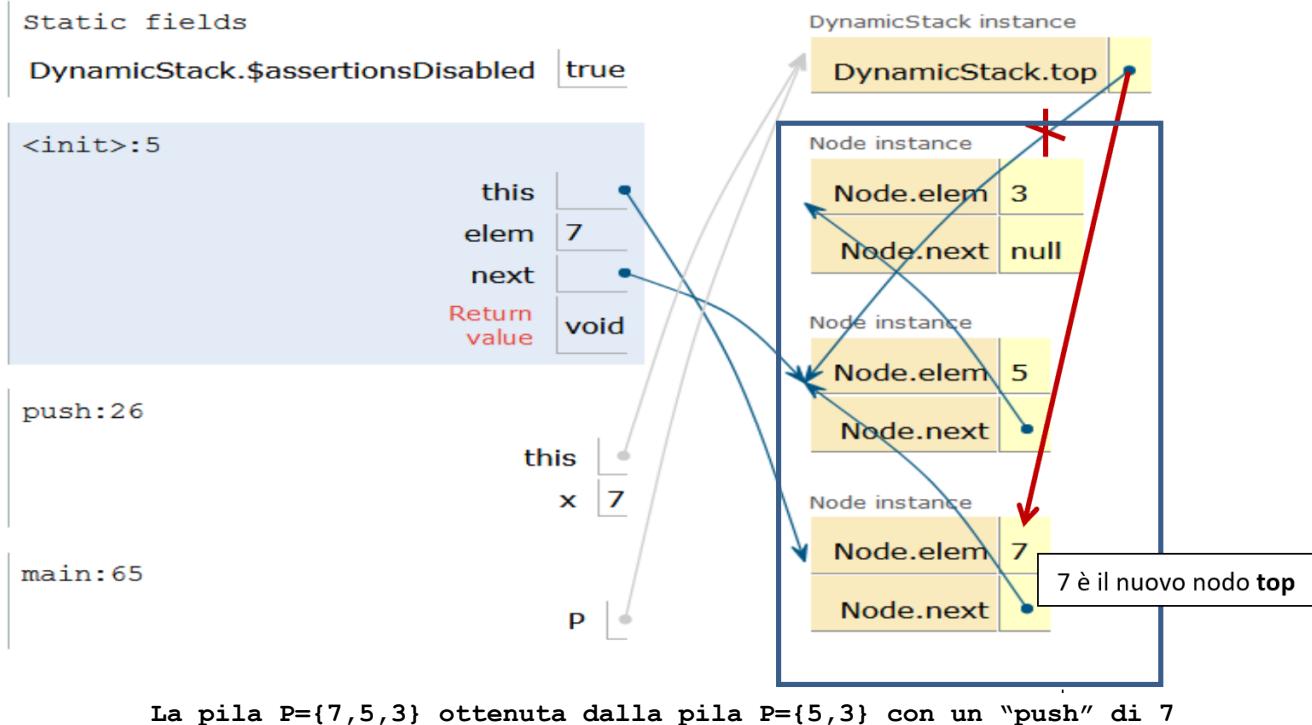
La classe delle pile ha: una operazione **void push(int x)** che aggiunge un nodo di elemento x "in cima" a una pila (oppure "a sinistra" se scriviamo la pila da sinistra a destra); una operazione **int pop()** che toglie il nodo in cima (o a sinistra) di una pila non vuota e ne restituisce l'elemento; **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={7,5,3}** (qui disegnata con la cima contenente **7** in basso). Una variabile di tipo pila contiene l'indirizzo di **top**, che ha sua volta è **l'indirizzo del primo nodo**. Quindi ogni nodo tranne l'ultimo contiene un elemento e l'indirizzo del nodo successivo.

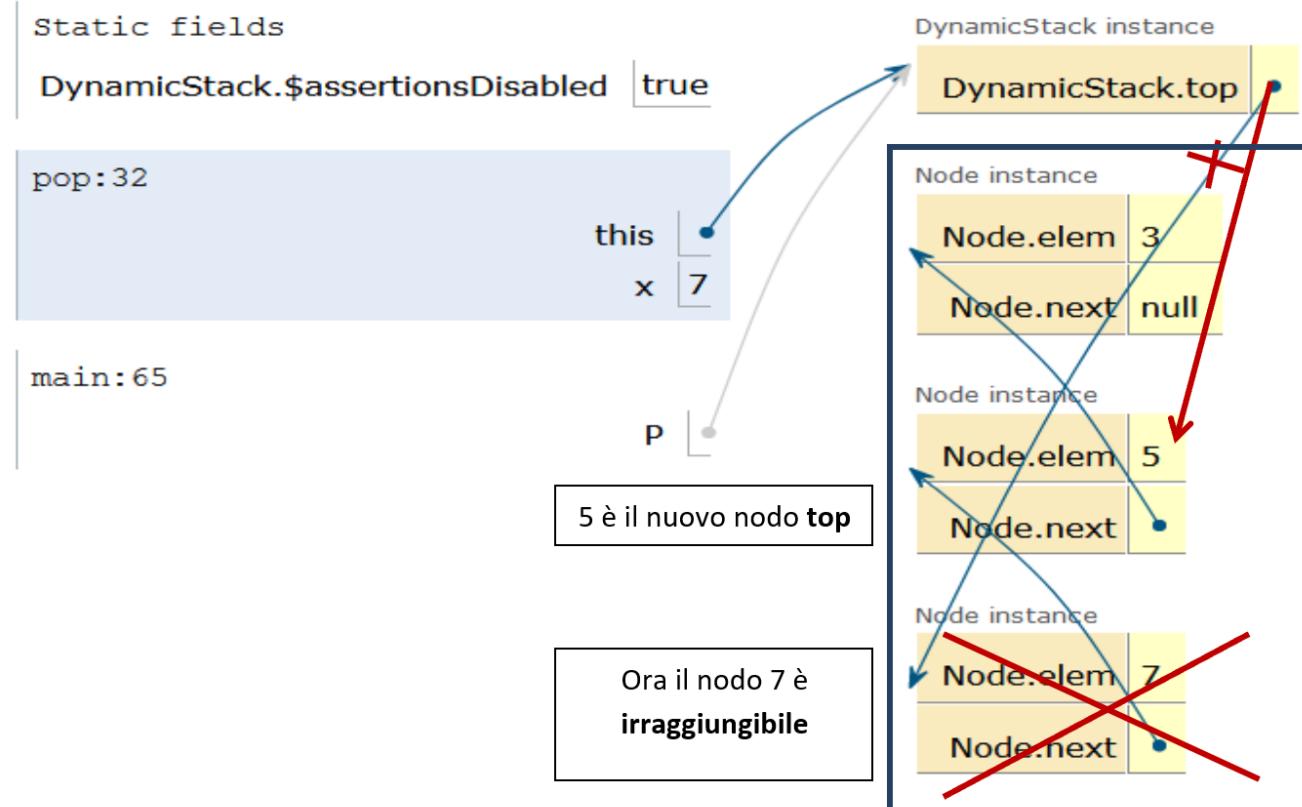


Vediamo come ottenere la pila **P={7,5,3}** a partire dalla pila **P={5,3}** aggiungendo 7 con un **push**.

Nel prossimo disegno, vediamo il nuovo nodo contenente 7 che punta al nodo contenente 5, che era il vecchio valore di "top". "top", che prima puntava al nodo contenente 5, viene ora riassegnato al nodo contenente 7.



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.



Realizziamo ora la classe ***DynamicStack*** (dopo averla implementata con gli array). È la struttura dati Stack (dunque segue la convenzione "Last In First Out") realizzata con gli oggetti di tipo Node. Le sue funzionalità dal punto di vista di un codice client che la utilizza rimangono le stesse, cambia l'implementazione interna, che però rimane invisibile al client.

Tutti i metodi di DynamicStack devono preservare il seguente ***invariante della classe***: ogni nodo tranne il primo punta all'elemento precedente, e top punta al primo elemento della pila. Come già visto per gli array estendibili, non consentiamo nessun accesso diretto ai nodi della pila: ogni accesso ai dati nella pila avviene con le operazioni pubbliche 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 Node
che parte da top e procede lungo la pila fino a arrivare al nodo
null. Usiamo di nuovo una conversione Node-->String. */
public String toString(){
    Node temp = top;      //partiamo dal nodo in cima alla pila
    String s = "";        //accumuliamo gli elementi in s
    while (temp != null){ //ci fermiamo quando temp arriva al nodo null
        s=s+" "+temp.getElem()+"\n"; //aggiungiamo l'elemento in cima
        temp=temp.getNext(); //avanziamo al nodo successivo
    }
    return s;
}
/* NOTA: dobbiamo salvare top in temp. Se avessimo usato top al posto
di temp, scrivendo top=top.getNext(), avremmo cancellato l'indirizzo
della cima della pila, e quindi perso l'accesso alla pila dopo
l'esecuzione del metodo. */

//COSTRUTTORE di una pila P = {1,...,n}, pila vuota se n<=0.
//Aggiunge i nodi nell'ordine da n fino a 1. 1 sta nel top.
public DynamicStack(int n){
    top = null; int i = n;
    while (i>=1) //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={11,9,7,5,3}" );
    DynamicStack P = new DynamicStack();
    P.push(3);P.push(5);P.push(7);P.push(9);P.push(11);
    System.out.println(P);
    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={5,3}" );
System.out.println(P);
System.out.println("Stampo la pila Q={1,2,3,4,5,6,7,8,9,10}");
DynamicStack Q = new DynamicStack(10);
System.out.println(Q);
}
}
```

Esercitazione 02

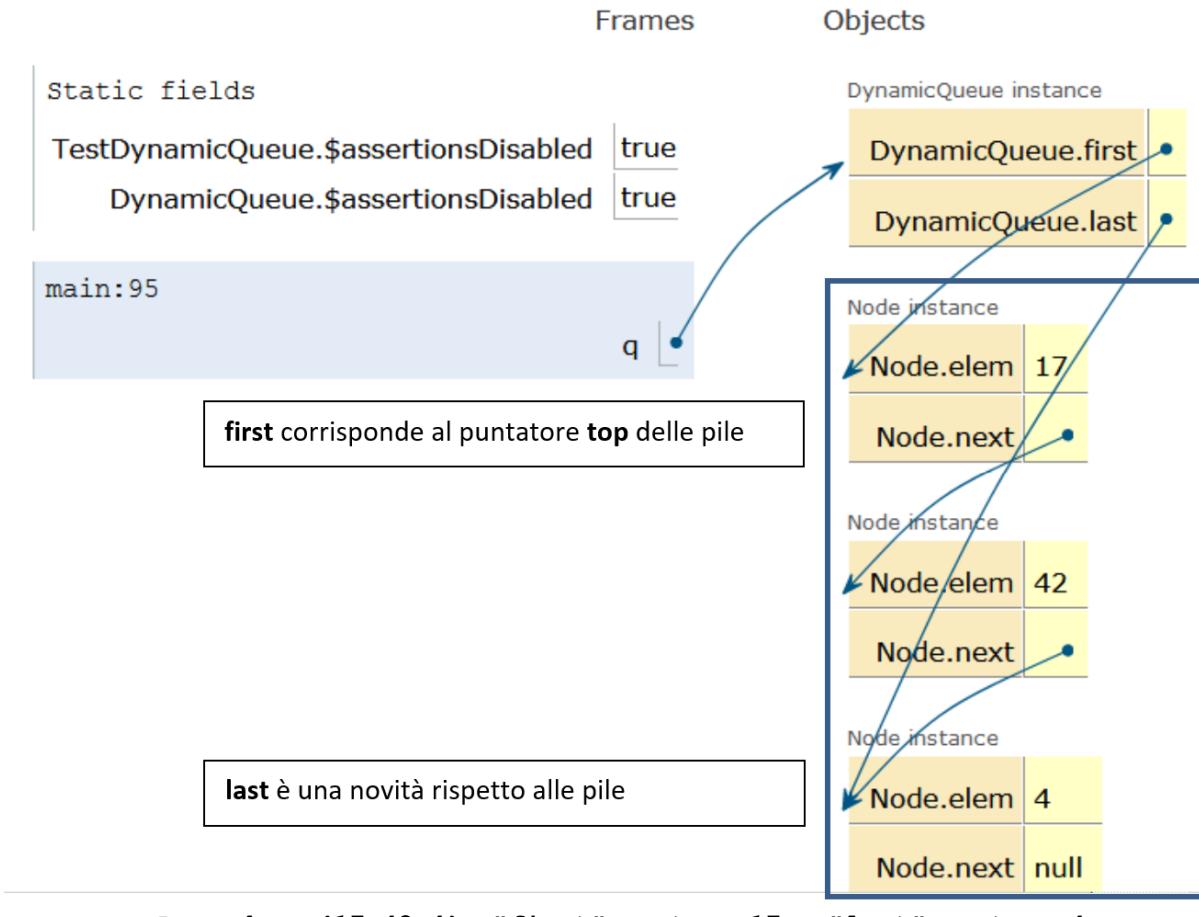
Code dinamiche

Una coda è una struttura dati 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 successivo (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, che punta alla fine della coda.

Vediamo il disegno una coda `{17, 42, 4}`, con il nodo "first" che contiene 17 in alto e con il nodo "last" che contiene 4 in basso. Una oggetto di tipo coda punta a una coppia di indirizzi first, last, muovendoci a partire da first nell'esempio troviamo 17 e l'indirizzo del nodo che contiene 42, in quest'ultimo l'indirizzo del nodo che contiene 4 e null (la fine della lista). Possiamo arrivare a 4 in un passo solo se seguiamo il puntatore last.



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. A differenza delle pile e code definite tramite array, i cicli per scorrere la struttura non usano indici interi, ma i puntatori/indirizzi, che vengono spostati all'elemento successivo (si veda il codice). **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 successivo, e first e last puntano al primo e all'ultimo elemento della coda. Inoltre first e last sono uguali a **null** se la coda è vuota.

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

Vediamo come cambia una coda $q=\{17, 42\}$ se aggiungiamo 4 in fondo. Il puntatore `first` non cambia, il puntatore `last` che puntava al nodo che contiene 42 ora punta al nodo che contiene 4. Anche il nodo contenente 42, che puntava a null ed era alla fine della coda, ora punta al nodo che contiene 4.

Static fields

`DynamicQueue.$assertionsDisabled` true

`<init>:5`

this	4
elem	4
next	null
Return value	void

`enqueue:32`

this	4
x	4

`main:80`

q	nuovo valore <code>last</code>
---	--------------------------------

DynamicQueue instance

`DynamicQueue.first`

`DynamicQueue.last`

Node instance

`Node.elem` 17

`Node.next`

Node instance

`Node.elem` 42

`Node.next`

Node instance

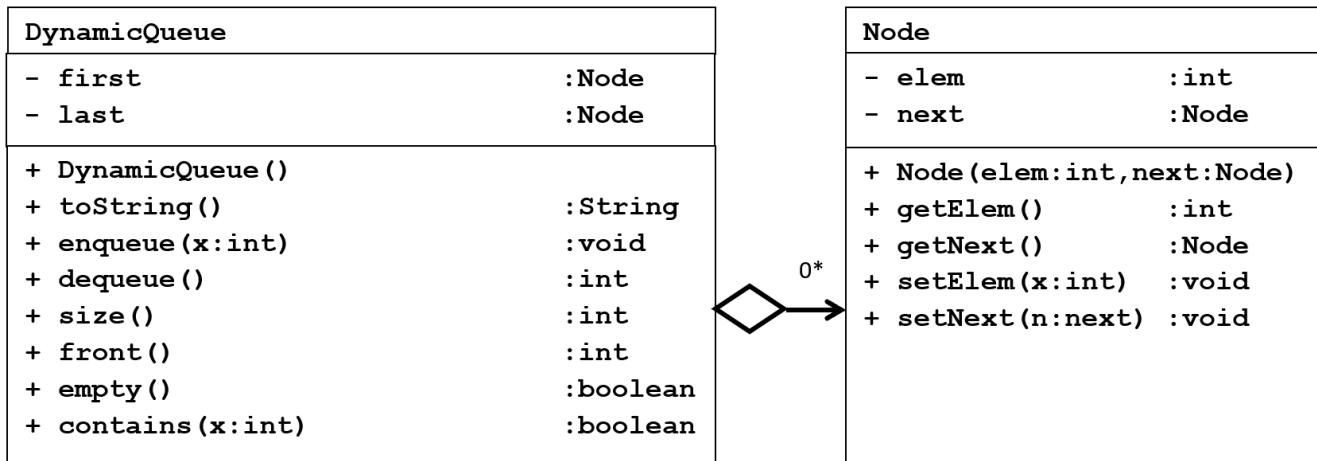
`Node.elem` 4

`Node.next` null

nuovo successore per il vecchio valore di `last`

Diagramma UML per le code dinamiche

Nel diagramma indichiamo che 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);
        System.out.print(q);
        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 2021

```

//DinamicQueue.java
public class DynamicQueue {
    private Node first;
    private Node last;
    public DynamicQueue(){
        first = last = null;
    }
    //Conversione DynamicQueue --> String
    public String toString(){
        Node temp = first; //partiamo dal primo nodo della coda
        String s = "";
        while (temp != null){
            //ci fermiamo quando temp arriva al nodo null
            s = s + " || " + temp.getElem() + "\n";
            temp=temp.getNext(); // arretriamo al nodo precedente }
        return s;
    }

    // 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=first;
    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 l'oggetto di tipo Node da cui partiamo per percorrerla, nodo che chiamiamo **il nodo in cima** alla lista (oppure: il nodo più a sinistra). Chiamiamo il nodo a cui punta un nodo dato **il nodo successivo** 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 gli array. Le differenze riguardano l'uso di indirizzi al posto di indici. Per esempio, possiamo dire che: **(i)** usiamo **null** al posto della posizione di indice **lunghezza del vettore** (quella a destra dell'ultima posizione esistente); **(ii)** usiamo il nodo in cima alla lista al posto della posizione **0** (quella più a sinistra); **(iii)** usiamo un nodo p (dunque un indirizzo) al posto di un indice i per indicare una posizione della lista; **(iv)** usiamo l'assegnazione p = p.getNext() al posto dell'assegnazione i=i+1 per ottenere la posizione successiva a una posizione data nella lista.

Qui sotto vediamo la lista `l={10,20,30,40,30,20,10}`, identificata con il puntatore al primo nodo, quello che contiene il primo 10. Ogni nodo contiene un puntatore al successivo: dal primo 10 si passa a 20, poi a 30 eccetera. Vediamo un metodo di stampa `scriviOutput()` che scorre la lista l, usando un puntatore p che parte da l e a ogni passo viene aggiornato a `p.getNext()`.



In questa immagine `scriviOutput(p)` ha un puntatore `p` al secondo nodo con un 30 in `l`, ha appena stampato 30, e si prepara a rimpiazzare `p` con un puntatore al prossimo nodo da stampare, contenente 20

Scegliamo di definire i metodi sulle liste di nodi ***fuori dalla classe Node***, di conseguenza questi metodi non possono venire inviati a oggetti della classe `Node`, quindi devono essere metodi statici con parametri di tipo `Node`.

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. Nel seguito, supponiamo che `q={10,20,30,40}` sia una lista.

0. **void scriviOutput(Node p)**. Stampa una lista concatenata partendo dal nodo in cima alla lista andando indietro (quindi in ordine inverso). Adattate il metodo per stampare una pila della Lezione 08. *scriviOutput(q) scrive: 10 20 30 40 andando a capo dopo ogni elemento.*
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. *length(q) vale 4.*
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. *sum(q) vale 10+20+30+40=100.*
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, mantenendo in una variabile m il più grande degli elementi trovati. Alla fine della lista m è il massimo. *max(q) vale 40.*
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. *member(q,30) vale true e member(q,50) vale false.*
5. **String toString(Node p)**. Restituisce una stringa con i nodi di p separati da uno spazio. *toString(q) vale "10 20 30 40".*
6. **boolean sorted(Node p)**. Verifica se una lista concatenata è ordinata in modo debolmente crescente. *sorted(q) vale true, se p = {10,20,30,40,30,20,10} allora sorted(p) vale false.*
7. **boolean equals(Node p, Node q)**. Verifica se due liste concatenate sono uguali: hanno gli stessi elementi nello stesso ordine. *equals(q,q) vale true, se p = {10,20,30,40,30,20,10} allora equals(p,q) vale false.*

```
//Node.java: riutilizziamo la classe già 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( "Main di prova per gli esercizi 0-7");
    System.out.println( "-----");

//aggiungo i nodi di q={10,20,30,40} a sinistra, dunque parto da 40
    Node q=new Node(40,null); q=new Node(30,q);q=new Node(20,q);
    q=new Node(10,q);
    System.out.println( "Lista q:");
    scriviOutput(q);
    System.out.println( "-----");

//aggiungo i nodi di p={10,20,30,40,30,20,10} a sinistra
    Node p=new Node(10,null); p=new Node(20,p); p=new Node(30,p);
    p=new Node(40,p); p=new Node(30,p); p=new Node(20,p);
    p=new Node(10,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. max_rec(p) = " + max_rec(p));
    System.out.println( "-----");
    System.out.println( "4. member(p,30) = " + member(p,30));
    System.out.println( "4. member(p,50) = " + member(p,50));
    System.out.println( "4. member_rec(p,30) = "+ member_rec(p,30));
    System.out.println( "4. member_rec(p,50) = "+ member_rec(p,50));
    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 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.

Un'avvertenza generale sull'uso delle variabili temporanee. Tutti i metodi di NodeUtil sono statici, quindi ricevono una copia p dell'indirizzo che definisce il nodo p, non l'originale di p. Queste copie sono **liberamente modificabili** senza modificare gli originali: per esempio in **scriviOutput(Node p)** posso scrivere **p=p.getNext()** senza modificare l'indirizzo originale di p. In questo caso **non serve** introdurre **una variabile temporanea** Node temp = p per salvare p. Naturalmente se invece uso un metodo set, per esempio p.setNext(r), modifico parte del p originario.

```
//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.getElement();
        p=p.getNext();
    }
    return s;
}

//versione ricorsiva
public static int sum_rec(Node p){
    if (p==null) return 0;
    else return p.getElement() + 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.getElement();
    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.getElement());
        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.getElement();
    else return Math.max(p.getElement(),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 separati da spazi
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 (secondo elemento > primo 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:
    //1. se p,q sono lo stesso indirizzo, allora p=q=null e p,q
    //contenevano lo stesso numero di elementi uguali a due a due:
    //erano uguali
    //2. se p,q sono indirizzi diversi, allora uno e' null e l'altro no
    //quindi p,q avevano lunghezze diverse:
    //erano diversi
    return (p==q);    //in ogni caso e' la risposta giusta
}

///////////////////////////////
//          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 (30 minuti). Molte costruzioni in Java vengono ripetute uguali per tipi diversi. Un esempio: per ogni tipo T, S (con T, S classi, non tipi primitivi) possiamo definire una classe i cui oggetti sono coppie di un oggetto di tipo T e un oggetto di tipo S. Altri esempi. 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 tipo (ricordiamo che in Java una classe definisce un tipo). Molte classi di libreria Java usano i generici. Per capire come usare i generici, vediamo ora un semplice esempio: 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=Integer 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. Dobbiamo anche evitare tutti i metodi che contengono riferimenti a caratteristiche particolari di T0, S0: il codice che resta deve poter funzionare con due classi **qualsiasi** T, S. In una classe generica posso utilizzare metodi che esistono in tutte le classi, per esempio: **boolean equals(T x)**, **String toString()**.

Attenti infine a **non scrivere GenericPair**, omettendo di precisare il valore delle variabili di classe T, S. Se lo fate, allora Java identifica *GenericPair* con **GenericPair<Object, Object>**, dove Object è la classe di tutti gli oggetti. **Questo identificazione spesso cancella dell'informazione essenziale**, e di conseguenza il programma non funziona bene. Scrivete invece:

GenericPair<T,S>, GenericPair<Integer,String>, ..., a seconda di quale classi T,S volete scegliere. Nei compilatori più recenti, è consentita invece **l'abbreviazione GenericPair<>**, purché Java sia in grado di dedurre in modo unico le istanziazioni dei tipi T,S.

```
//GenericPair.java
```

```
/* Permette di creare oggetti "coppia di elementi (x,y)" di tipo
rispettivamente T e S, tipi non noti a priori. Sarà il codice client
che al momento in cui eseguiamo una new istanzierà le variabili di
classe con delle classi (tipi) specifiche (trovate degli esempi nel
file TestGenericPair.java). La classe coppia viene detta una classe
parametrica, i suoi parametri sono i parametri di tipo T e S. Si dice
anche che è una classe polimorfa, perché può assumere più ruoli a
seconda della scelta di T e S, favorendo il riuso del codice.
```

```
*/
```

```
public class GenericPair<T,S> {
    private T first; private S second;
    // T, S parametri di tipo che rappresentano variabili di
    // classe (e dunque dei tipi)
    // T e S NON possono essere tipi primitivi
    // (limitazione solo apparente, viene superata introducendo il
    // concetto di "autoboxing" che vedremo tra poco)
```

```
/* 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 classe T, S
quindi puo' essere usato (se le classi sostituite a T e S non
definiscono il toString() viene usato quella della classe Object, che
restituisce la stringa indirizzo-nella-heap in esadecimale) */
public String toString()
    {return "(" + first.toString() + "," + second.toString() + ")";}
```

```
}
```

Come esempio di metodo **statico generico** definiamo fuori dalla classe GenericPair<T,S> un metodo che dati i tipi T,S e una coppia in GenericPair<T,S> inverte le componenti della coppia, e restituisce un risultato in GenericPair<S,T>. Il metodo costruisce una nuova coppia e non altera quella originaria. Quando una definizione avviene fuori da GenericPair<T,S>, dove T,S non sono dichiarate come variabili di classe, allora **dobbiamo dichiarare che T,S sono variabili di classe prima del tipo di ritorno, scrivendo <T,S>**. Se un metodo 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:

Siamo fuori da GenericPair<T,S> e dobbiamo indicare che T,S sono variabili di classe scrivendo <T,S>

```
public static <T,S> Classe<T,S> metodo(...)
```

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

    /* Questo metodo, dato un (x,y) di tipo GenericPair, restituisce un nuovo (y,x) di tipo GenericPair. Attenzione qui sotto a non scrivere GenericPair al posto di GenericPair<S,T> come tipo di ritorno, altrimenti il compilatore e' costretto a indovinare S, T, e rimpiazza GenericPair con GenericPair<Object,Object>, perdendo informazioni. */

    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){
        /* Qui creiamo un GenericPair<T,S> in cui il primo elemento (campo first) e' di tipo T=String e il secondo (campo second) e' di tipo S=Integer. Al posto di GenericPair<String,Integer> dopo la new posso scrivere GenericPair<>, purche' Java sia in grado di dedurre in modo unico i valori di S, T. Attenzione che i compilatori vecchi non accettano la seconda versione.*/
        GenericPair<String,Integer> p = new GenericPair<>("pluto", 1);
        // nel seguito, "\t" e' la tabulazione e inserisce spazi bianchi
        System.out.println( "p = \t\t\t\t" + p);
        System.out.println( "inv(p) = \t\t\t\t" + inv(p));
        System.out.println( "p non cambia: \t\t" + p);
        //Dato che inv e' un metodo statico, al di fuori della sua classe
```

```

//deve venire chiamato p. es. come:
// TestGenericPair.<String, Integer>inv(p);
}
}

```

Lezione 10. Parte 2: nodi generici e tipo Integer (60 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>.

Attenti solo a **non scrivere GenericNode**, omettendo di precisare il valore della variabile di classe T. Il motivo è lo stesso già visto con GenericPair: Java identifica **GenericNode** con **GenericNode<Object>**, ma di solito questa identificazione cancella dell'informazione essenziale per il programma. È consentita invece **l'abbreviazione GenericNode<>**, purché Java sia in grado di dedurre in modo unico il valore di T.

Autoboxing: i tipi Integer, Boolean e Double. Un'altra 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 che punta a un intero, booleano e reale, ovvero a un oggetto che contiene un intero, booleano e reale. Immaginiamo l'indirizzo di un intero, booleano, reale come una **box ("scatola")** che contiene un intero, booleano, reale. Le classi Integer, Boolean, Double sono fatte di queste "scatole", e sono versioni **equivalenti** degli interi, booleani, reali, possiamo scrivere una al posto dell'altra senza problemi. Se necessario, Java trasforma automaticamente un int in un Integer e viceversa con operazioni dette **autoboxing (inscatolamento automatico)**. Boxing indica l'operazione di mettere, per esempio, un int in un Integer, unboxing il viceversa.

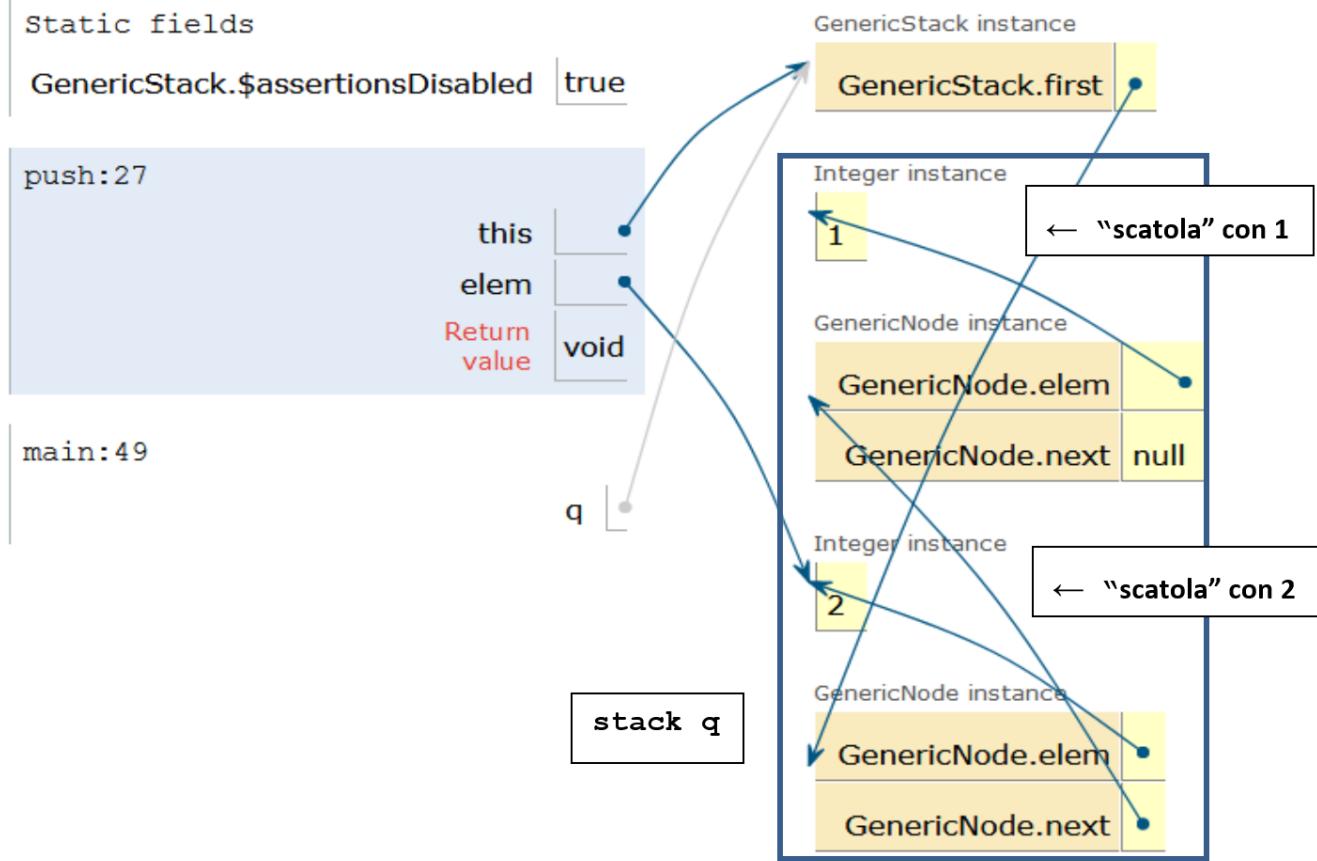


Immagine della lista `q={2,1}` (per la descrizione vedi sotto)

Nel disegno qui sopra vediamo come esempio una lista `q = {2,1}` i cui elementi hanno tipo Integer anziché int. La lista `q` è un puntatore al primo dei due nodi, che contiene l'indirizzo di 2 (chiamato "integer instances" e non intero), disegnato come una freccia che punta a una **"scatola"** attorno a 2. Il nodo con 2 contiene anche l'indirizzo di un nodo che contiene 1 e fatto allo stesso modo: il nodo contiene una freccia che punta a 2, non il numero 2. L'immagine si riferisce all'istante in cui inseriamo 2 eseguendo `q.push(2)`.

```
// GenericNode.java
public class GenericNode<T> {
/* Attenzione qui sotto a non scrivere GenericNode al posto di
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(T elem){this.elem=elem;}
public void setNext(GenericNode<T> next){this.next=next;}
}

```

A partire dalla classe `GenericNode<T>`, nodi con elementi di un tipo `T`, con `T arbitrario`, possiamo definire una classe `GenericStack<T>` per rappresentare stack di elementi di tipo `T`, con `T arbitrario`. La costruzione è la stessa usata per definire la classe `DynamicStack` delle pile dinamiche `su int` a partire dalla classe `Node` dei nodi `su int`.

```

import java.util.*; //Importiamo tutte le Java utilities

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

    public GenericStack(){first = null;}

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

    /* Al posto di GenericNode<T> posso scrivere GenericNode< >,
    purche' Java sia in grado di dedurre in modo unico il valore T */
    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 String toString(){
        GenericNode<T> p = first; String s = "";
        while(p!=null) {
            s = s + p.getElem() + " ";
            p=p.getNext();
        }
        return s;
    }
}

```

```

}
}
```

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.*;
//Importiamo tutte le Java utilities
//inclusa la classe Random dei generatori di numeri casuali

public class TestGenericStack {

    public static void main(String[] args){

        // Creiamo uno stack per contenere stringhe
        System.out.println( " ---> Stampo p = {\\"hello \\", \\"world!\\"} " );
        /* Al posto di GenericStack<String> dopo la new posso scrivere
        GenericStack<>, se il compilatore ha informazione sufficiente per
        inferire il tipo String. Attenzione però: i compilatori piu' vecchi
        non accettano la versione GenericStack<> che qui impieghiamo. */
        GenericStack<String> p = new GenericStack<>(); //p pila String
        p.push("world!");
        // OK: il metodo push si aspetta un argomento di tipo String
        p.push("hello ");
        System.out.println(p); // stampo 2 strighe
        String s1 = p.pop();
        // OK: il metodo pop ritorna un valore di tipo String
        String s2 = p.pop();
        p.push(s1 + s2); // OK: s1 + s2 produce una nuova stringa
        System.out.println( " ---> Stampo p = {\\"hello world!\\"} " );
        System.out.println(p);

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

        // Creiamo uno stack per contenere numeri interi.
        // 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 = {2,1} " );
        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);
System.out.println(q); // stampo 2, 1 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 (+) */
System.out.println(" ---> Stampo q = {2+1} ");
System.out.println(q); // stampo 3 intero

// 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());

/* Il metodo toString() ci fornisce una versione stampabile per
Stack di elementi di tipo arbitrario (ma tutti dello stesso tipo)
*/
System.out.println(" ---> ora p e' uno stack di 1 stringa");
System.out.println(p); // OK: p e' uno Stack di String
System.out.println(" ---> ora q e' uno stack di 1 Integer");
System.out.println(q); // OK: q e' uno Stack di Integer
System.out.println(" ---> s e' uno stack di " + n + " Double");
System.out.println(s); // 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, ma per il resto riutilizzando i metodi di C per quanto è possibile. Questo sia allo scopo di **ridurre il lavoro**, sia allo scopo di **evitare la moltiplicazione degli errori**, quasi inevitabile quando si moltiplica del codice simile. Inoltre riutilizzando il codice di C lo ricontrolliamo e lo correggiamo, e ci troviamo ad utilizzare codice più sicuro. 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 e 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 String toString() //conversione bottiglia --> stringa
{return " " + livello + "/" + capacita;}
}

```

Estensione di una classe. Aggiungiamo un attributo alla classe Bottiglia dotandola di un tappo rappresentato da una variabile booleana. Il tappo può essere in due stati (aperto o chiuso) e vogliamo fare in modo che il versamento di liquido dalla e nella bottiglia abbia effetto solo quando la bottiglia è aperta.

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 C ma non nella sottoclasse D. Si noti che un oggetto di BottigliaConTappo è anche un oggetto Bottiglia, quindi un oggetto BottigliaConTappo si può usare ovunque si possa usare un oggetto Bottiglia, ma non vale il viceversa.

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 il corpo di un metodo (**fare override**), 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 hanno successo con una bottiglia aperta, ma falliscono con una bottiglia chiusa.

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" ("sopraclass") indichiamo la classe C che viene estesa da D. Un costruttore della classe estesa D **deve iniziare con l'invocazione "super(...)"** a un costruttore della sopraclass C, altrimenti Java chiama comunque il costruttore di default "super()" di C. L'obbligatorietà è data dal fatto che la responsabilità di inzializzare i campi della sopraclass è della sopraclass stessa. In un metodo sovrascritto **è possibile** chiamare la versione dello stesso metodo della sopraclass con sintassi "**super.metodo(...)**".

```
// 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 superiore per fare le
         inizializzazioni della capacita' */
        super(capacita);
        // supponiamo che la bottiglia sia inizialmente chiusa
        aperta = false;
    }
    /* La chiamata al costruttore della classe superiore deve essere la
     prima istruzione del costruttore della sottoclasse. Il costruttore
     della classe superiore può essere richiamato solo dal costruttore della
     sottoclasse. Se il costruttore della sottoclasse non richiama
     esplicitamente un costruttore della classe superiore, per prima cosa
     viene chiamato automaticamente il costruttore predefinito della classe
     superiore, quello senza parametri (super()); se la classe superiore
     non ha un costruttore senza parametri il compilatore genera un errore.
    */
}
```

```
// 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 (ma non toString()) 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); /*super.aggiungi() indica il metodo
"aggiungi" nella classe Bottiglia che stiamo estendendo */
    else return 0;
}

/* OVERRIDE del metodo "rimuovi" per versare liquido dalla bottiglia:
stesse osservazioni */
public int rimuovi(int quantita){
    if (aperta)
        return super.rimuovi(quantita); /*super.rimuovi() indica il metodo
"rimuovi" nella classe Bottiglia che stiamo estendendo */
    else return 0;
}

/* OVERRIDE del metodo "toString()". Alla stringa che descrive una
bottiglia aggiungiamo l'informazione aperta/chiusa */
public String toString(){
    return super.toString() + " (aperta = " + aperta + ")";
}

// BottigliaConTappoDemo.java
/* Controlliamo che lo stato "bottiglia aperta" lasci invariati i
```

```

travasi, e che lo stato "bottiglia chiusa" li azzera. */
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();
        System.out.println(b);
        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);
        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(b);

        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);
        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.

Innanzitutto, osserviamo che **assert** e **if** hanno funzioni diverse. Vediamo quando usare l'uno e l'altro.

L'**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.

L'**if** serve invece a controllare il flusso 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):" sqrt(" + x + ") ";
    return (int) Math.sqrt((double) x);
}
```

Qui il programmatore ha definito la funzione "radice quadrata" per numeri interi in termini dell'analogia 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): "push su pila piena: size = " + size;
    data[size++] = x; //Vuol dire: prima data[size]=x; poi size++;
}
```

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. Queste situazioni si gestiscono meglio usando il concetto di **eccezione**, che vedremo più avanti.

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];
//Vuol dire: prima --size; poi 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.

Ripetiamo, 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.**

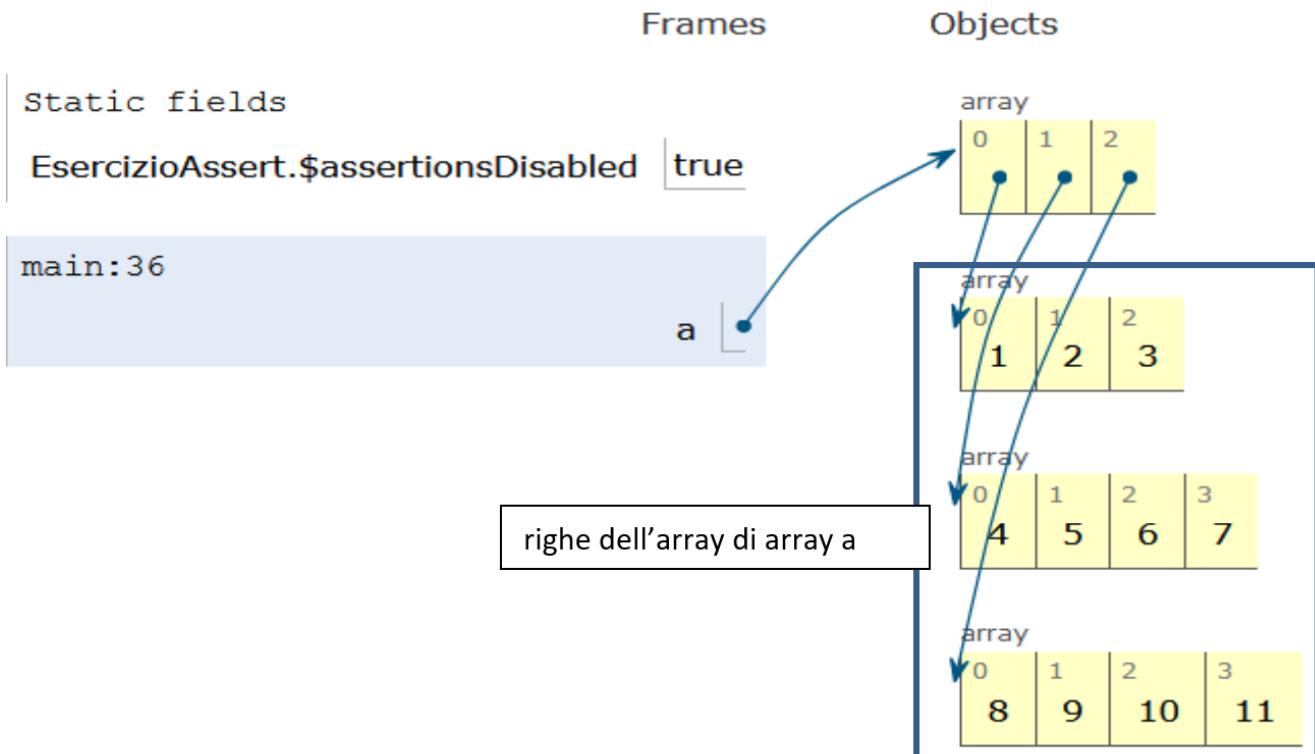
Definiremo l'asserzione per il metodo usando un metodo ausiliario:

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

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

Cercate di immaginare cosa succede al metodo quando viene applicato all'array di array descritto e disegnato qui sotto.

Prima un breve ripasso sugli array di array. L'array di array `a` è l'indirizzo di un vettore, ognuno dei quali è l'indirizzo di un vettore-riga. Nel vettore-riga troviamo degli interi, le righe non hanno sempre la stessa lunghezza. Per raggiungere l'indirizzo della riga `i` dobbiamo scrivere `a[i]`, per ottenere l'elemento di posto `j` nella riga `a[i]` dobbiamo scrivere `a[i][j]`. Nel prossimo disegno vediamo un vettore `a` composto di tre righe, la prima riga con `{1,2,3}`, la seconda con `{4,5,6,7}`, la terza con `{8,9,10,11}`.



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 = array di array-riga di interi. Per esempio una riga di tre
elementi, seguita da una riga di due, seguita da una riga di quattro:
```

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 ArrayOutOfBoundsException
```

```

    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
        ( " ok(a) =      " + ok(a) +"\n 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}}
    System.out.println( " ok(b) =      " + ok(b) );
    // 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. Il codice di un programma viene di solito scritto attraverso tante modifiche successive. Una modifica può consistere nell'estensione di una classe. In questa lezione, vediamo cosa succede quando estendiamo ripetutamente una classe in Java.

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. Questo modificatore di visibilità consente di utilizzare top in ogni classe nello stesso package (nella stessa cartella) e in ogni classe che estende la classe data (anche in altre cartelle), ma in nessun altro caso. Nel nostro caso dobbiamo raggiungere la posizione top di una pila quando ne calcoliamo il massimo in un'altra classe. Questa visibilità ci impedisce di modificare un attributo di una classe di una cartella che importiamo (cosa che comunque di solito è meglio non fare, per evitare la propagazione di errori da una classe all'altra), ma rende più facile la definizione di classi nella stessa cartella.

Estendere comporta aggiungere un'invariante di classe con delle condizioni che descrivano quali valori vogliamo avere in 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"
// Cancellate il costruttore DynamicStack(int n). Otterrete:

public class DynamicStack {
    protected Node top;
    /* Rispetto alla classe Dynamic Stack, cambia la visibilità di top
       da private a protected: visibile a tutte le classi dello stesso
       package, e a tutte le sottoclassi, anche in package diversi. Questo
       perché top dovrà essere accessibile da parte dei metodi delle
       sottoclassi di DynamicStack che definiremo. */
}

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 String toString(){
    Node temp = top; String s = "";
    while (temp!=null){
        s = s + " || " + temp.getElem() + "\n";
        temp = temp.getNext();
    }
    return s;
}

}

//DynamicStackMax.java
public class DynamicStackMax extends DynamicStack {
// Manteniamo il campo 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 oppure si trovano nella stessa cartella.

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 subito dopo 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 conversione in stringa
public String toString(){
    return super.toString() + " || max= " + max + "\n";
}
}

```

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
    /* Se il costruttore della sottoclasse non richiama esplicitamente
       un costruttore della classe superiore, per prima cosa viene
       chiamato automaticamente il costruttore predefinito della classe
       superiore, cioe' quello senza parametri (super()); se la classe
       superiore non ha un costruttore senza parametri il compilatore genera
       un errore. */
    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 conversione in stringa
public String toString(){
    return super.toString() + " || size = " + size + "\n";
}

// 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);
        System.out.print(P);

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

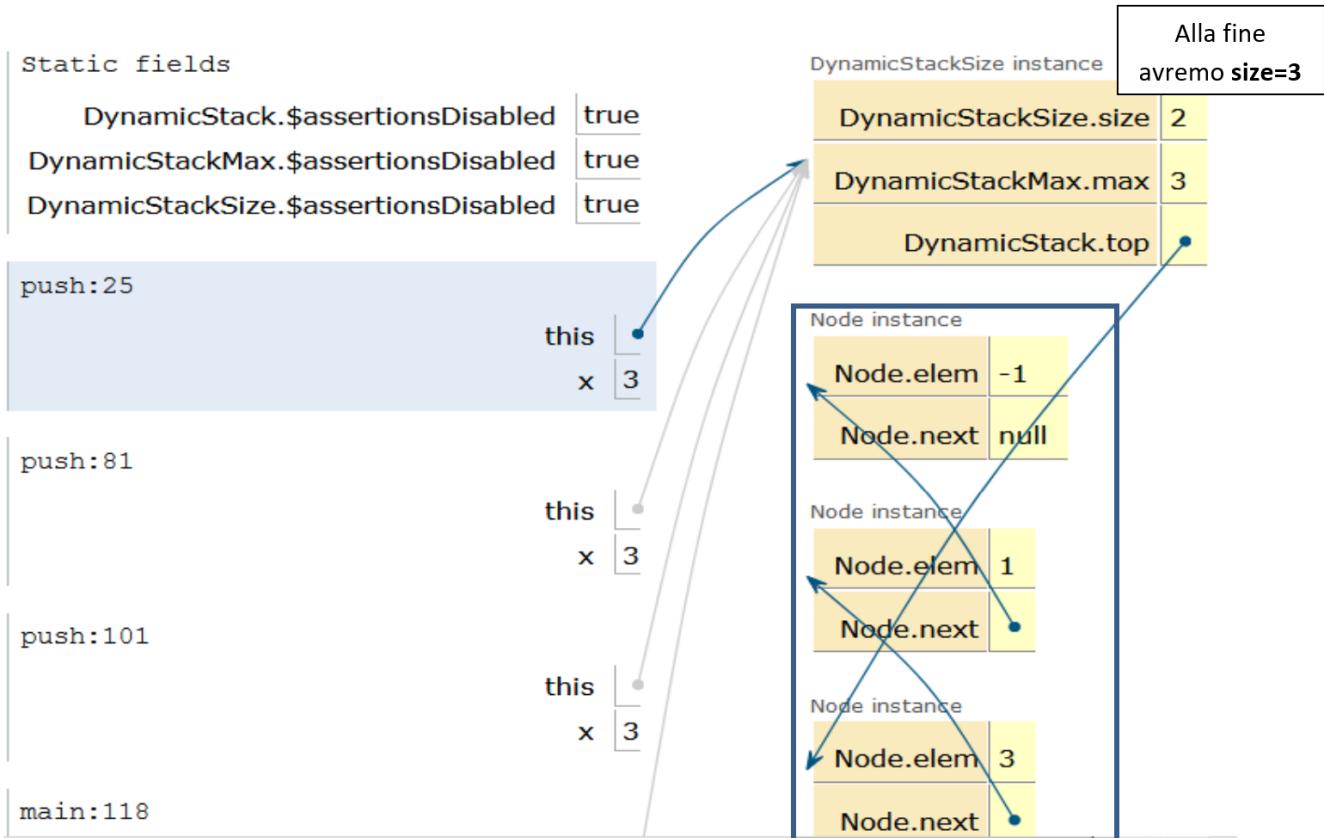
        System.out.println( "Estraggo 11, 9, 7. Leggo 5" );
        System.out.println( " P.pop() = " + P.pop());
        System.out.println( " P.pop() = " + P.pop());
        System.out.println( " P.pop() = " + P.pop());
        //Leggiamo il prossimo elemento, 5, senza estrarlo dalla pila
        System.out.println(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

```

Un esempio di calcolo su una pila P di DynamicStackSize

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.

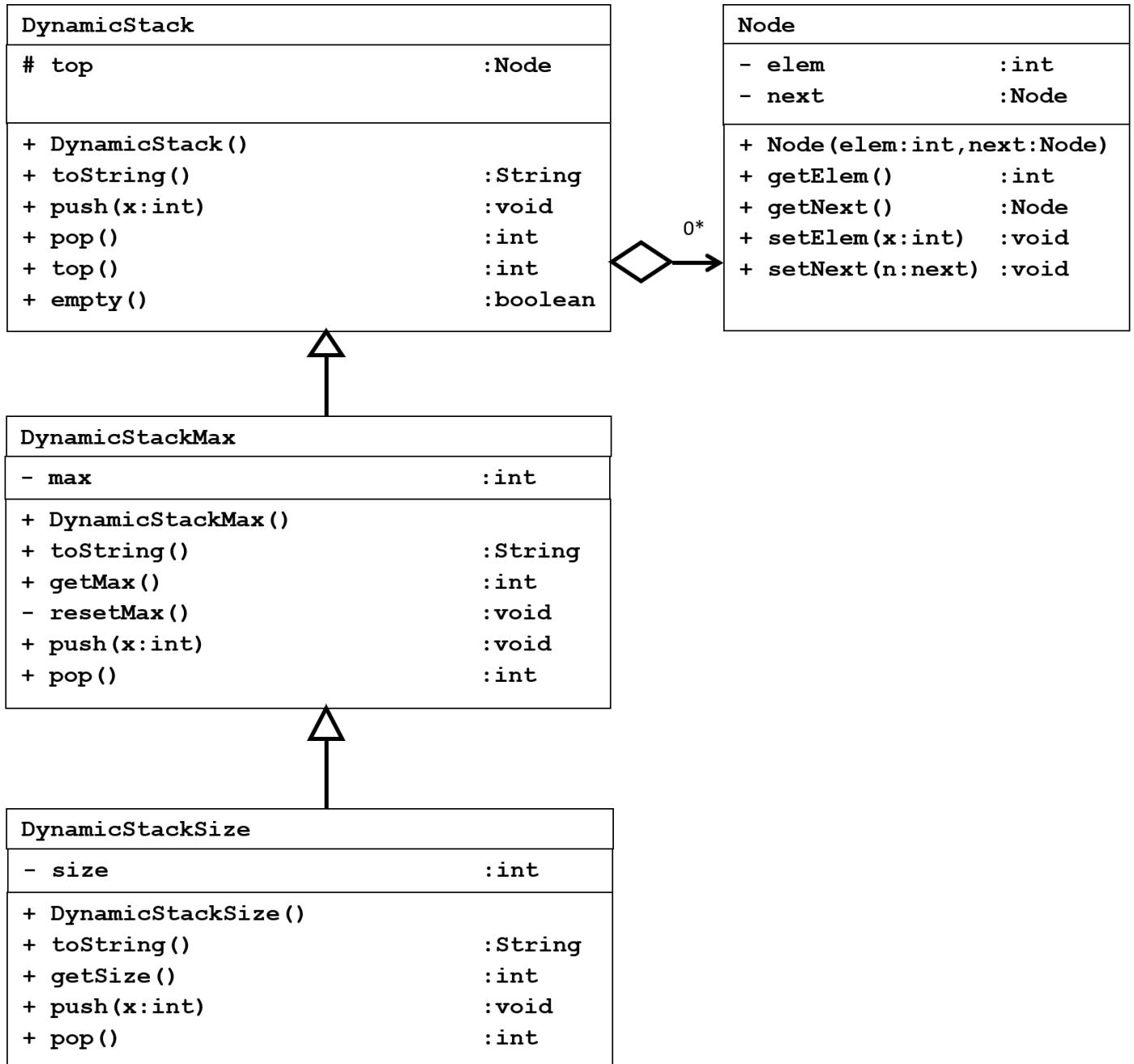
Nel prossimo disegno vediamo la situazione dello Stack + Heap in questo istante.



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 questa osservazione definiamo il diagramma UML per DynamicStack e Node. Le estensioni **DynamickStackMax** e **DynamickStackSize** si indicano con una **freccia verso l'alto** dalla classe che estende verso la superclasse. In un diagramma UML, un attributo **protected** ("top" nel nostro caso) si indica con #.



Lezione 13

Tipo esatto e binding dinamico

Lezione 13. Parte 1. Tipo esatto e tipo apparente. Downcast e upcast, binding dinamico. Supponiamo che C sia una classe che estende una classe D, ovvero che “class C extends 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 probabilmente degli attributi specifici di C. Basta ignorare gli attributi specifici di C (senza per questo cancellarli), per considerare l’oggetto di tipo C come un oggetto di tipo D. Questa operazione è disponibile in Java e si chiama **upcasting**. Grazie all’upcasting, un oggetto in Java può avere diversi tipi, tutte estensioni dello stesso tipo. Chiamiamo **tipo esatto** di un oggetto il tipo con cui l’oggetto è stato costruito tramite una new(): allora l’oggetto può avere come tipo tutti e soli i tipi che includono (cioè che estendono) il suo tipo esatto.

L’operazione opposta (in una certa misura) di **upcast** si chiama **downcast**. Un downcast di obj alla classe C si indica con `((C) obj)`. Durante la compilazione, `((C) obj)` con obj di **tipo apparente D** viene considerato di tipo C se questo è almeno possibile, cioè se esiste una sottoclasse **E inclusa in C, D**. Altrimenti il compilatore dà errore. Occorre però poi che anche durante l’esecuzione obj abbia davvero un **tipo esatto E incluso in C,D**. Altrimenti calcolare `((C) obj)` solleva una eccezione, detta **ClassCastException**, e termina il programma, dato che non è possibile considerare obj di tipo C. Un downcast `((C) obj)` andrebbe utilizzato solo quando si è **assolutamente certi** che, in tutte le esecuzioni possibili, obj ha tipo esatto C o incluso in C. Di solito il downcast si usa per passare a una classe **più piccola**, dal tipo apparente D a un tipo C incluso in D. Tuttavia vedremo esempi di downcast corretti **con la classe C non inclusa in D**, ma con una sottoclasse E in comune con D.

Si noti che l’upcast è invece sempre valido, perché è corretto sostituire un oggetto di un tipo più preciso al posto di un tipo meno preciso. Per esempio, se “class C extends D”, gli oggetti di tipo C possono essere usati in tutti i contesti in cui si usano gli oggetti di tipo D (ma non vale il viceversa), perché un oggetto di tipo C può fare tutto quello che può fare (ovvero rispondere ai metodi di) un oggetto di tipo D.

In ogni comando Java si chiama **tipo apparente** di una espressione il tipo che questa espressione ha in base al tipo delle variabili che la compongono. 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 dà un errore di tipo (**non compila**). In questo corso faremo particolare attenzione ad imparare come si evitano gli errori di tipo.

Se il programma compila, durante l'esecuzione Java conosce i valori delle variabili di una espressione, e da esse deduce il tipo esatto di un oggetto. Java utilizza il tipo esatto di un oggetto obj per **decidere quale versione di un metodo applicare all'oggetto (ovvero da quale classe nella gerarchia prendiamo il metodo)**. Questo meccanismo viene chiamato **dynamic binding** (perché avviene a **run time**) o anche, a volte, **late binding** (perché avviene **dopo la compilazione**), ma la seconda terminologia è meno usata perché si applica anche in altri ambiti dei linguaggi di programmazione. Il dynamic binding è un argomento centrale del corso. Se Java deve chiamare 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, quella contenuta nel tipo apparente di obj. A volte ce ne sono anche altre, e prevale la prima trovata.

Un esempio. Vediamo un esempio di una variabile di tipo Bottiglia che, a seconda delle circostanze dell'esecuzione del programma, può avere tipo esatto la classe Bottiglia oppure tipo esatto la classe BottigliaConTappo, più precisa (ovvero di tipo *più piccolo*).

```
//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 la circostanza "oggi_piove" e' vera o
    // falsa, il tipo esatto di b e' la sottoclasse oppure la classe.
```

```

boolean oggi_piove = true;
//boolean oggi_piove = false;
// Provate entrambe le possibilita' qui sopra: una non compila

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

// DOWNCAST: di solito, 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 downcast 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 sottoclassse 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 due bottiglie `a10` e `b10`, con tipi originari **BottigliaConTappo** e **Bottiglia**. Facciamo vedere che `a10` è una istanza delle classi **BottigliaConTappo** e **Bottiglia**, mentre `b10` solo di **Bottiglia**, e `null` non è istanza di nessuna delle due classi (anche se `null` ha tipo sia **BottigliaConTappo** che **Bottiglia**).

```
public class InstanceOfDemo{

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

    public static void main(String[] args){
        //a10 e b10 sono oggetti instanziati dal costruttore, quindi !=null:
        Bottiglia a10=new BottigliaConTappo(10);
        Bottiglia b10=new Bottiglia(10);
        BottigliaConTappo u = null;

        System.out.println("a10 instanceof BottigliaConTappo = " +
                           (a10 instanceof BottigliaConTappo)); // true
        System.out.println("b10 instanceof BottigliaConTappo = " +
                           (b10 instanceof BottigliaConTappo)); // false
        System.out.println("u=null instanceof BottigliaConTappo = " +
                           (u instanceof BottigliaConTappo)); // false
        System.out.println();
        System.out.println("a10 instanceof Bottiglia = " +
                           (a10 instanceof Bottiglia)); // true
        System.out.println("b10 instanceof Bottiglia = " +
                           (b10 instanceof Bottiglia)); // true
        System.out.println("u=null instanceof Bottiglia = " +
                           (u instanceof Bottiglia)); // false
    }
}
```

Esempi ulteriori sui concetti di sottotipo, upcast, downcast, tipo apparente e tipo vero.

Il concetto di **sottotipo** permette di dare più di un tipo a un valore, rendendo quindi più flessibile l'utilizzo di tale valore. Partiamo da

un esempio che già conoscete, e consideriamo il tipo int e il tipo double di Java. Questi due tipi definiscono due insiemi di valori. L'insieme dei numeri rappresentati in int è un sottoinsieme dell'insieme dei numeri rappresentati in double, perché un intero, in particolare, è un reale con la parte decimale uguale a 0. Non è vero il viceversa, cioè un reale non è necessariamente un intero (perché non tutti i double hanno parte decimale pari a 0).

Si dice, in questo caso, che "int è sottotipo di double" e si può usare la notazione:

```
int <: double
```

Informalmente, possiamo dire che int "è più preciso" di double, perché di un int si sa che la parte decimale di sicuro è 0.

L'assegnazione:

```
int x = 10;
double y = x;
```

è corretta, perché un int si può considerare un double. Questo cambiamento di tipo si dice **upcast** (quando un oggetto si sposta da un tipo più piccolo a un tipo più grande). Un upcast è sempre accettato dal compilatore, perché matematicamente corretto.

Un assegnamento di natura opposta:

```
double k = 3.2;
int y = k;
```

invece NON è corretto, perché si avrebbe perdita di informazione: si perderebbe la parte decimale di k. Java permette questa perdita di informazione se il programmatore la indica esplicitamente con un'operatore di cast, che in questo caso è detto **downcast** (spostamento da un tipo più grande a un tipo più piccolo):

```
double k = 3.2;
int y = (int)k;
```

Sappiamo che la definizione di una classe Java **definisce un nuovo tipo** (infatti il nome di una classe può essere il tipo di una variabile, di un parametro, di un valore di ritorno di un metodo). Quando si introduce una sottoclasse con una "extends" si introduce anche un sottotipo:

```
BottigliaConTappo extends Bottiglia {...}
```

Implica, con la notazione appena vista, che:

```
BottigliaConTappo <: Bottiglia
```

Vale lo stesso discorso di upcast che per i tipi primitivi, ovvero posso scrivere:

```
Bottiglia b2 = new BottigliaConTappo(10);
```

assegnando un oggetto di un tipo più piccolo (più specifico) un oggetto di tipo più grande. Infatti tutte gli oggetti che appartengono all'insieme BottigliaConTappo sono anche Bottiglie (ma non è vero il viceversa).

Si parla di "tipo apparente" per indicare il tipo con cui viene *dichiarata* una variabile, si dice "tipo vero" quello indotto dalla 'new', quando l'oggetto viene costruito. Per esempio:

```
(1) BottigliaConTappo b1 = new BottigliaConTappo(10);
```

```
(2) Bottiglia b2 = new BottigliaConTappo(10);
```

in (1) tipo apparente e tipo vero coincidono, mentre invece in (2) il tipo apparente è Bottiglia, mentre il tipo vero è BottigliaConTappo.

Il compilatore fa i controlli di tipo sui tipi apparenti, più restrittivi (quindi che garantiscono una sicurezza maggiore), ma durante l'esecuzione del programma, a runtime, la versione del metodo da chiamare dipende dal tipo vero. Per esempio, se scriviamo:

```
int q = b2.aggiungi(3);
```

la versione del metodo aggiungi() che verrà invocata sarà quella della (sotto)classe BottigliaConTappo, che sovrascrive (fa l'override del) metodo corrispondente della (sopra)classe Bottiglia, specializzandolo. Anche il risultato di b2.aggiungi(3) cambia: nel caso di una bottiglia con tappo, l'aggiunta di 3 litri fallisce se la bottiglia è chiusa.

Anche fra classi è possibile fare downcast esplicativi, ovvero:

```
BottigliaConTappo c1 = new BottigliaConTappo(3);
```

```
Bottiglia c2 = c1;
```

```
BottigliaConTappo c3 = (BottigliaConTappo)c2; // downcast
```

L'ultimo assegnamento dice al compilatore di considerare un oggetto di tipo apparente Bottiglia (c2) come BottigliaConTappo. Cosa succede al runtime? Va tutto bene, perché il tipo vero di c2 è BottigliaConTappo. Invece ci sarebbe un errore a runtime se il tipo vero NON fosse BottigliaConTappo (o una sua sottoclasse).

È sempre possibile fare un test con 'instanceof' per scoprire il tipo vero (corrispondente alla classe di un oggetto) prima di un downcast (vedere Lezione 13), per evitare errori a runtime:

```
if (c2 instanceof BottigliaConTappo)
```

```
// caso particolare: c2 == null --> instanceof false
```

```
BottigliaConTappo c3 = (BottigliaConTappo)c2; // downcast
```

Si noti che:

```
c1.aperta(); // compila
c2.aperta(); // NON compila
c3.aperta(); // compila
```

perché il compilatore controlla se il tipo apparente di c2 corrisponde a una classe che contiene il metodo aperta() e questo è solo contenuto nella classe BottigliaConTappo.

Con il programma che segue, facciamo qualche esperimento con instanceof, cast, binding dinamico.

Ci ricordiamo che:

- il compilatore fa i controlli di tipo usando i tipi *apparenti*;
- a runtime, viene scelta la versione di un metodo (dinamico) in base al tipo *vero* dell'oggetto su cui viene chiamato il metodo.

Ricordiamo anche che un downcast (C)obj funziona a runtime solo se:

- obj ha tipo vero C;
- obj ha tipo vero D <: C.

Infine, ricordiamo il comportamento di **instanceof**:

```
(obj instanceof C) = true
    SE obj=/=null AND tipo_vero(obj) = C
(obj instanceof C) = false
ALTRIMENTI.
```

```
public class Prova {

    public static void prova(Bottiglia b){
        // questo metodo accetta come parametri
        // oggetti di tipo Bottiglia o di tipo
        // più piccolo (per es. BottigliaConTappo)

        System.out.println( "b.aggiungi(6) =" + b.aggiungi(6));
        // tutto ok, b ha tipo Bottiglia quindi posso
        // chiamare aggiungi()

        // instanceof controlla il tipo *vero*
        // di un oggetto:
        if (b instanceof Bottiglia)
            System.out.println( "b è una bottiglia" );
    }
}
```

```

if (b instanceof BottigliaConTappo) {
    System.out.println( "b è una bottiglia con tappo" );
    // boolean r1 = b.aperta(); // non compila
    // perché b ha tipo apparente Bottiglia
    // che non ha un metodo aperta()
    BottigliaConTappo z = (BottigliaConTappo)b;
    // questo downcast funziona a runtime perché
    // il tipo vero di b è BottigliaConTappo (certificato
    // dalla instanceof)
    System.out.println( " z.aperta()=" + z.aperta()); // compila
    // perché z ha tipo apparente BottigliaConTappo
    System.out.println( " b.aggiungi(6)=" +b.aggiungi(6)+" b='"+b);
    System.out.println( " z.aggiungi(3)=" +z.aggiungi(3)+" z='"+z);
    // tutte e due le precedenti chiamate a aggiungi()
    // chiamano la versione di BottigliaConTappo di
    // aggiungi(): il loro tipo vero è infatti
    // BottigliaConTappo grazie al binding dinamico!
}
}

public static void main(String[] args){
    Bottiglia a = new Bottiglia(9);
    System.out.println( "\nProvo con un Bottiglia =" + a);
    prova(a);
    BottigliaConTappo b = new BottigliaConTappo(9);
    System.out.println( "\nProvo con un BottigliaConTappo =" + b);
    prova(b);
    /* Quest'ultima chiamata a prova() stampa:
       b è una bottiglia
       b è una bottiglia con tappo
       .....
       Perché?
    */
}
}

```

Nota avanzata. Riassumiamo come possiamo usare il downcast quando un metodo m ha come parametro formale un oggetto di un tipo (apparente) C, dove C è una classe genitore di una gerarchia. Al metodo m possiamo passare come parametri attuali oggetti delle sottoclassi di C, usando un upcast: un oggetto con tipo più preciso si può utilizzare al posto

di uno con tipo meno preciso. Nel corpo del metodo m possiamo voler scegliere che azioni fare a seconda se il parametro attuale è istanza oppure no di un tipo dato. Possiamo ottenere questo effetto usando "instanceof", come mostrato sopra.

Lezione 13. Parte 2. Un esempio di ereditarietà e di binding dinamico. Java contiene la classe **Graphics** degli oggetti grafici: un oggetto grafico è **un'area rettangolare dello schermo** in cui ho dei metodi per disegnare. Vediamo come programmare con la classe Graphics usando l'ereditarietà. Le tappe sono le seguenti.

1. A partire da **Graphics**, definiamo una classe **Figura** di figure, ciascuna con un suo metodo **draw(Graphics g)** che disegna la figura in un oggetto grafico g. Definiamo un metodo draw vuoto per una figura generica (disegniamo la figura vuota) quindi usando l'ereditarietà ri-definiamo draw in un modo non banale ogni volta che definiamo una sottoclasse di Figure.
2. A partire da Figura, definiamo la sottoclasse **Disegno** delle finestre Java (= oggetti della classe **Jframe**) con incluso un array di Figure. Noi forniamo un metodo **void paint(Graphics g)** che viene mandato a un array di figure, prende in ingresso in un oggetto grafico g posto nella finestra, e disegna tutte le figure dell'array in g usando draw.
3. Alla fine, invochiamo su un disegno un metodo di libreria **setVisible(true)**, che fornisce un oggetto grafico g al metodo paint per la finestra e genera il disegno. Non è possibile definire g noi stessi: il costruttore della classe Graphics è **protected**, dunque **inaccessibile in Disegno e Figura**, che non sono sottoclassi di Graphics e non fanno parte della stessa cartella.

Proviamo ora a usare l'ereditarietà e definire metodi draw per quadrati e cerchi. Presentiamo la classe Jframe delle finestre e definiamo la classe Figura.

```
// Graphics = classe del pacchetto awt, con gli oggetti grafici
// JFrame = classe del pacchetto swing, con "finestre" fornite di:
//           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 array di figure prese da
// classi diverse e di disegnarle usando un unico comando draw(),
// perché Figura è un sopra-tipo dei tipi delle figure delle
// classi Quadrato e Cerchio.
```

```
// Figura.java
import java.awt.*;      //Abstract Window Toolkit (finestre grafiche)
import javax.swing.*; //estensione di awt per interfacce 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
    // sottoclasse della classe Figura.
    public void draw(Graphics g){ /* disegna la figura vuota */ }
}
```

Per la sottoclasse “**Quadrato**” di “Figura”, il metodo draw disegna in g un quadrato di lato dato, orizzontale e centrato con gli assi (includiamo una figura alla fine della lezione). Usiamo i metodi della classe Graphics: **setColor**, **drawLine**, **drawOval**.

```
// Quadrato.java quadrato = una possibile forma di una Figura
// Definiamo Quadrato come una sottoclasse di Figura
import java.awt.*;      //Abstract Window Toolkit
import javax.swing.*; //estensione di awt per interfacce 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 (vuoto in Figura)
    // 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 in g
        g.drawLine(-m,   m, -m, -m);    //disegno secondo lato in g
        g.drawLine(-m, -m,   m, -m);    //disegno terzo lato in g
        g.drawLine( m, -m,   m,   m);    //disegno quarto lato in g
    }
}
```

Per la sottoclasse “**Cerchio**” di “Figura”, il metodo draw disegna un cerchio di raggio dato e centrato con gli assi (includiamo una figura alla fine della lezione).

```

// 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 interfacce 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 un oggetto grafico g, nel caso la figura sia un cerchio.
    // Disegniamo 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 interfacce grafiche

public class Disegno extends JFrame {
    /* Un "disegno" e' un JFrame con parte grafica = tutte le figure di
       un array 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 array di figure
    }
}

```

```

}

// OVERRIDE: ridefiniamo il metodo paint di JFrame
// chiedendo di inizializzare una finestra grafica, poi
// di disegnare tutte le figure dell'array figure di g
public void paint(Graphics g) { //INIZIALIZZO g
    int w = this.getSize().width; // base di g
    int h = this.getSize().height; // altezza di g
    g.clearRect(0, 0, w, h); // azzero il contenuto di g
    g.translate(w/2,h/2); //traslo sistema di riferimento al centro di g

    //DISEGNO tutte le figure dell'array 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];
    //Array di n figure: all'inizio ogni figura vale null
    //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 array di figure proprio "figure"
    Disegno frame = new Disegno(figure); //Jframe con array di figure

    //ESEMPI DI EREDITARIETA' (SENZA OVERRIDE) DALLA CLASSE JFRAME
    //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);
    // setVisible(true) rende il disegno visibile, inviando il metodo
    // paint all'oggetto frame insieme con un oggetto grafico g:
    //(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**

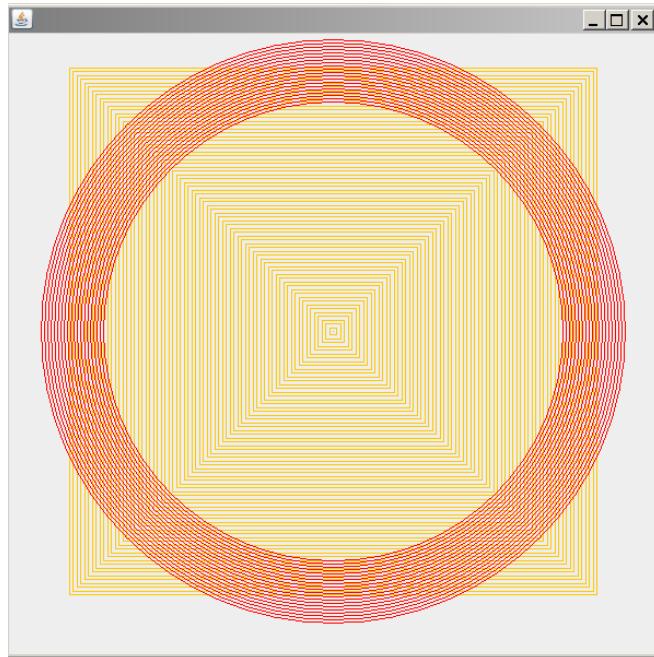
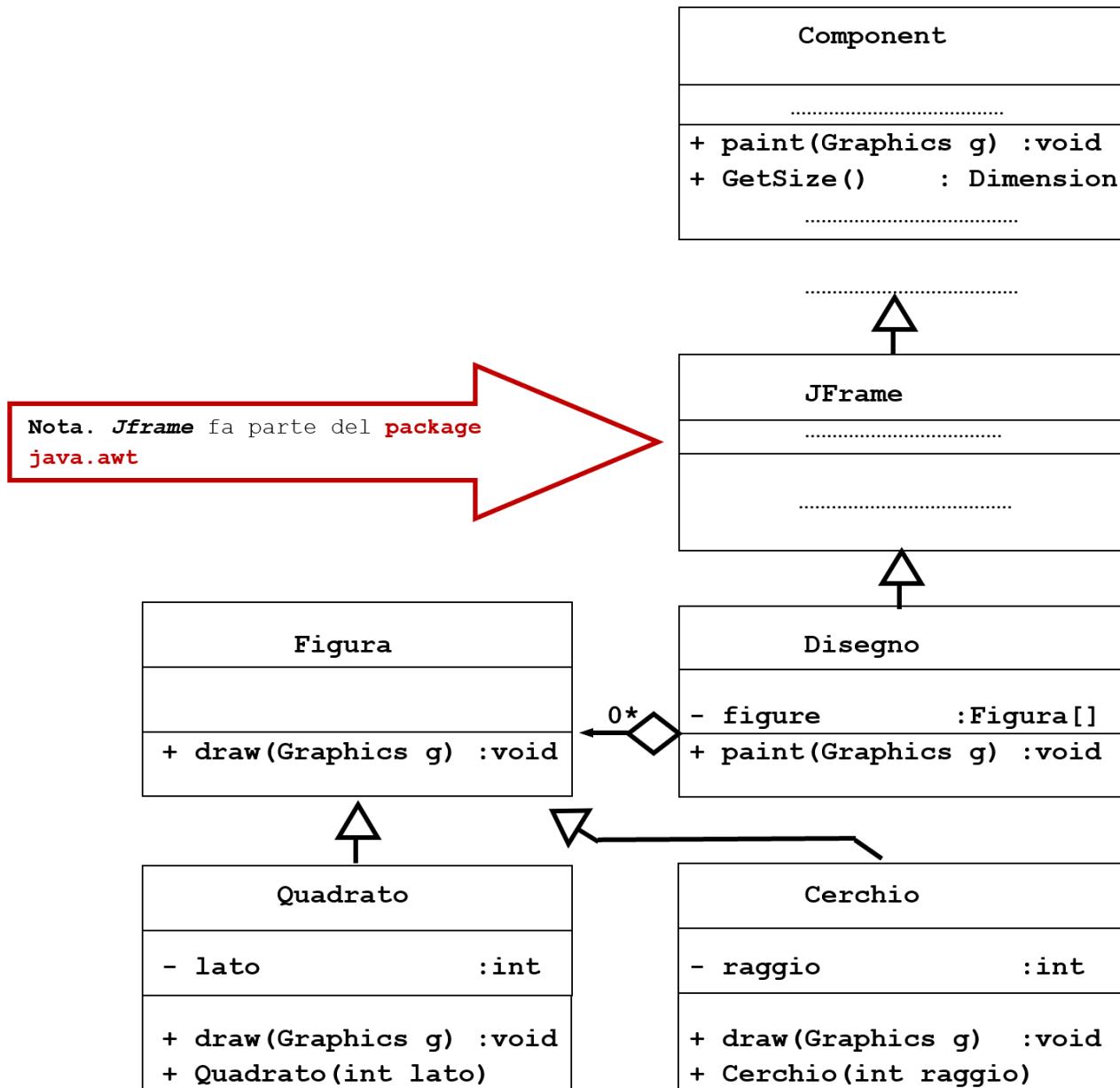
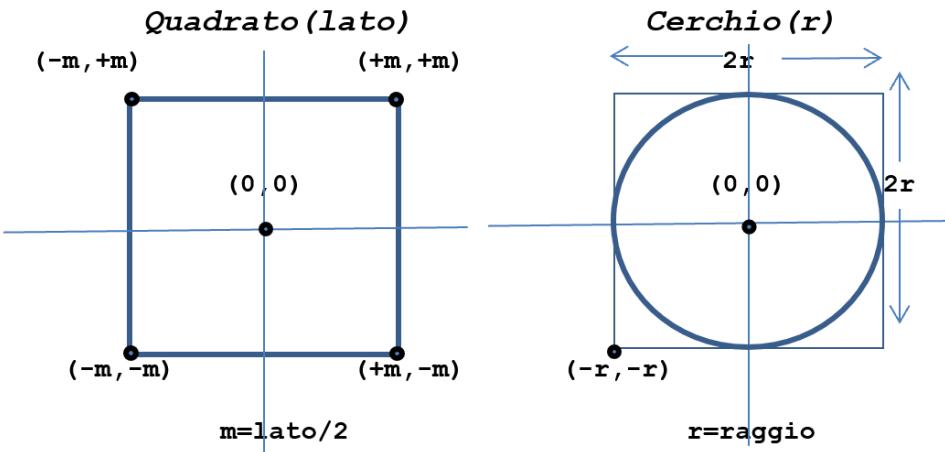


Diagramma UML della classe Disegno

Quadrato e Cerchio sono sottoclassi di Figura. La classe Disegno aggrega zero o più figure. Otteniamo Disegno come sottoclasse della classe di libreria JFrame delle finestre grafiche in Java, a sua volta ottenuta come sottoclasse di altre classi. Nel diagramma non inseriamo la lista completa delle sovracclassi di *JFrame*, classe importata da Java.



Posizione delle figure nel piano



Lezione 14

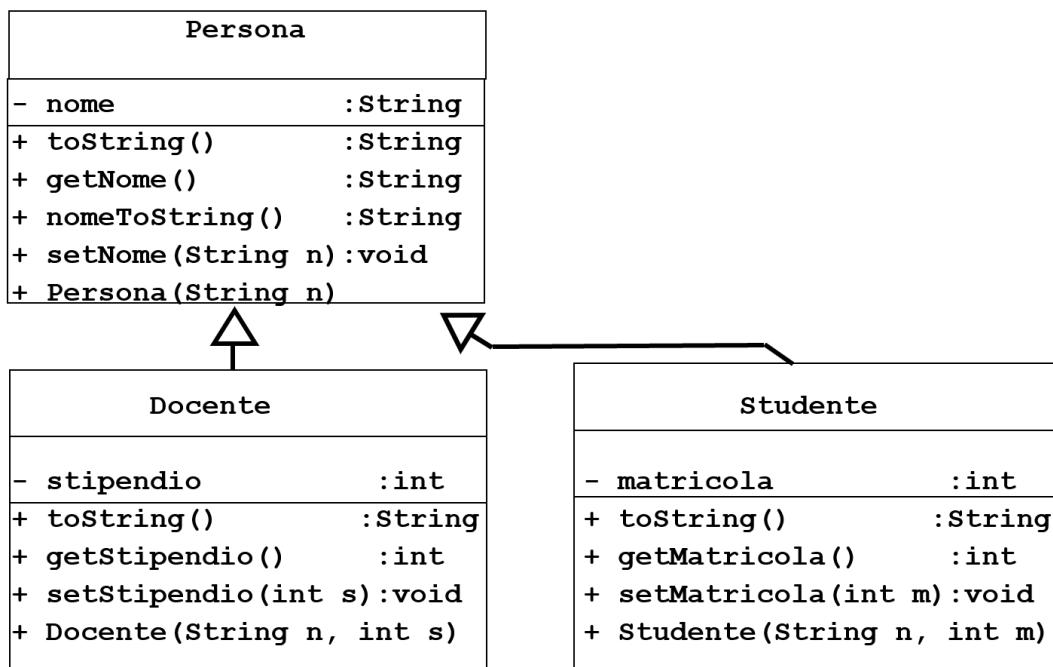
Esempi di ereditarietà e array come liste

Lezione 14. Parte 1. Vediamo nuovi esempi di uso del dynamic binding.

Nella lezione precedente, abbiamo visto una classe Figura di figure, con diversi metodi di disegno a seconda del tipo esatto della figura: il metodo da applicare veniva deciso con la regola del dynamic binding. Vediamo ora l'esempio di una classe **Persona** con due metodi uguali di conversione in stringa, **String toString()** e **String nomeToString()**. Riscriviamo **toString()** in ogni sottoclasse, non riscriviamo invece **nomeToString()**. Proveremo quindi ad applicare i due metodi a un array di persone. Nel caso di **toString()** il metodo usato dipende dalla sottoclasse a cui appartiene una data persona, nel caso di **nomeToString()**, invece, applichiamo sempre il metodo della classe **Persona**.

Cominciamo con un **diagramma UML della classe Persona** e le sue sottoclassi Docente e Studente. Nella classe **Persona** abbiamo un attributo **nome** con **get** e **set**, nelle classi **Docente** e **Studente** abbiamo un attributo in più, rispettivamente **stipendio** e **matricola**, con **get** e **set**.

Diagramma UML della classe Persona



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

    public String getNome(String nome) {
        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 String toString(){
        return " nome = " + nome;
    }

//Metodo di scrittura di cui NON intendo fare OVERRIDE:
//resta uguale anche se quando sono informazioni in piu'
    public String nomeToString(){
        return " 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 di toString(): raccolgo le informazioni in piu'
public String toString(){
    return super.toString() + " stipendio = " + stipendio;
}

//NON faccio override di nomeToString():
//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 toString(): raccolgo le informazioni in piu'
    public String toString(){
        return super.toString() + " matricola = " + matricola;
    }

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

```
}
```

Quando applichiamo ***String toString()*** e ***String nomeToString()*** a docenti e studenti nella classe Persona, in base alle regole del Dynamic Binding, viene usato il metodo definito nel tipo più vicino (nel diagramma UML) al tipo esatto. 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
        Persona  c = new Persona( "Bianchi");
        //Definisco un array v con le persone appena introdotte
        int n=3;
        Persona[] v = new Persona[n];
        v[0]=a;
        v[1]=b;
        v[2]=c;
        //tipo apparente v[0],v[1],v[2]: Persona, Persona, Persona
        //tipo esatto: Studente, Docente, Persona

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

        //Stampo c usando il metodo nomeToString() (SENZA override):
        //Java utilizza il tipo esatto di ogni oggetto e il metodo
        //nomeToString() per il tipo esatto. IN QUESTO CASO IL METODO E'
        //EREDITATO E RESTA SEMPRE LO STESSO (no override)
        System.out.println( "\nEsempio di nomeToString()" );
        for(int i=0;i<n;i++){
            System.out.println(i + " " + v[i].nomeToString());
        }
    }
}
```

```

    }
}
}
```

Lezione 14. Parte 2. Le classi MiniLinkedList e Iterator. Lasciamo momentaneamente da parte ereditarietà e dynamic binding per svolgere un esercizio sulle liste di nodi, in cui una classe viene usata come etichetta per consentire/proibire una certa operazione, in questo caso, la traversata veloce di una lista. Il nostro obiettivo è definire le classi **MiniLinkedList e Iterator** che implementano gli array tramite liste e l'iterazione su queste liste: questo comporta una difficoltà non immediatamente visibile.

Definiamo la classe **MiniLinkedList** delle liste concatenate V , attraverso una lista di nodi $V_0, \dots, V_{(size-1)}$, ognuno dei quali punta al successivo, e con un attributo **size** che indica il numero dei nodi. L'attributo **size** è solo un esempio: potremmo avere diversi attributi che descrivono caratteristiche importanti della lista. 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 $0 \leq i < size$. Con accesso lento, numero di `getNext()` richiesti = i .
2. **void set(int i, int x)** Assegna il valore x al nodo numero i , se $0 \leq i < size$.
3. **void add(int i, int x)** Aggiunge un nodo di contenuto x in posizione i , se $0 \leq i \leq size$, e incrementa $size$.
4. **int remove(int i)** che cancella il nodo di posto i dalla lista, se $0 \leq i < size$, e decrementa $size$.

Aggiungiamo un metodo privato **Node node(int i)** che restituisce l'indirizzo del nodo numero i . Viene usato dai metodi pubblici, non è pubblico perché consentirebbe un accesso ai nodi.

MiniLinkedList assomiglia alla classe degli array 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 $0, \dots, i-1$, quindi usa un numero di `getNext()` uguale ad i . L'accesso a un array statico, invece, avviene con un solo accesso a un indirizzo.

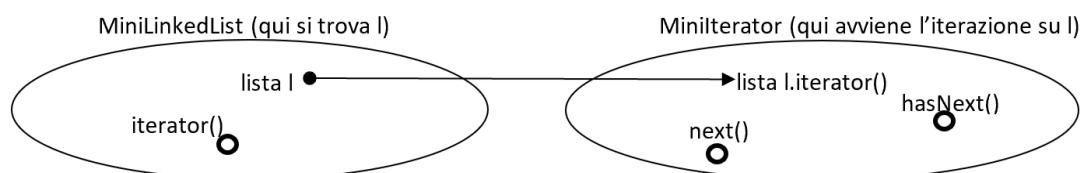
Spieghiamo ora come definire **l'iterazione** su MiniLinkedList: ora vediamo comparire la difficoltà. Supponiamo infatti che il nostro

obiettivo sia passare attraverso ciascun nodo della lista, **svolgendo per ognuno di essi la stessa operazione**. Questo compito richiede $0+1+2+\dots+(size-1)$ volte `getNext()`, uguale a $size(size-1)/2$ volte `getNext()`, un netto **svantaggio** per grandi valori di size rispetto al numero di operazioni che richiederebbe un array.

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 scrivere del codice che modifica la lista dall'esterno (ovvero senza utilizzare i metodi), vanificando l'invariante di classe "size rappresenta il numero di elementi presenti nella struttura", se, per esempio, il programmatore si dimentica di modificare l'attributo `size` della lista quando aggiunge/toglie nodi.

Per consentire una scansione veloce della lista ma senza consentire di modificare direttamente i puntatori ai nodi (ma solo con i metodi di `MiniLinkedList`) definiamo un metodo `iterator()`, che "racchiude" una lista `l` in un oggetto di una nuova classe, detta **MinIterator**, dentro alla quale avverrà la scansione. `MinIterator` ha un metodo **`int next()`** che legge il contenuto di un puntatore e sposta il puntatore al nodo dopo, ma **senza modificare i puntatori dei nodi**. L'oggetto `MinIterator` può essere usato per iterare la stessa operazione (per esempio: la stampa) su tutti gli elementi della lista `l`, senza consentire di aggiungere/togliere nodi da `l`. Per farlo, siamo obbligati a usare i metodi di **MinLinkedList**, che modificano l'attributo `size` se la dimensione della lista cambia, e ci impediscono di sbagliare il valore di `size`. Riassumiamo con il seguente disegno:



Realizziamo ora `MiniLinkedList` e `MinIterator` in Java.

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

```

//MiniLinkedList.java
public class MiniLinkedList
{
    private Node first; private int size;
    // INVARIANTE DI CLASSE: (first punta a 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
(-> per evitare information leak). */

private Node node(int i){
    //controllo che V_i sia un nodo della lista
    assert 0 <= i && i < size: "i non in [0,size-1]";
    //creo una copia di first per non modificare l'originale
    Node p = this.first;
    // il primo nodo lo consideriamo in posizione 0
    while (i > 0){
        //rimpiazzo per i volte il p con il nodo dopo
        assert p != null: "size non rispetta invariante";
        //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: "size non rispetta invariante";
    //se vale l'invariante questo assert e' vero
    return p;
    //nel caso i=0, node(i) restituisce proprio p = first
}

```

```

}

/** 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 = il nodo da eliminare =
    //= nodo i-esimo =
    //= nodo che viene dopo prev (= nodo (i-1)-esimo):
    Node el = prev.getNext();
    x = el.getElement();
    //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
// 1 piu' il numero di elementi della lista. Dobbiamo sottrarre 1:
size--;
return x;
}

/* Definiamo un metodo che inserisce una MiniLinkedList in un oggetto
che fa parte della classe MiniIterator. MiniIterator consente di
eseguire un ciclo senza rendere pubblici gli indirizzi dei nodi. */
public MiniIterator iterator(){
    return new MiniIterator(first);
}
// Per visitare la lista l scriverò MiniIterator it = l.iterator()
// e farò muovere it, si veda sotto nella classe TestMiniIterator.
}

/* MiniIterator.java - Classe che consente di traversare una volta sola
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;
    }
}

```

```

/* next() restituisce l'elemento nel nodo corrente e muove il
puntatore next al nodo dopo. Si noti che next() cancella il valore
originale di next: la visita della lista l viene fatta una volta sola.
Per fare un'altra visita dovrò creare nel codice client un
altro oggetto it = l.iterator(). */

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
        //Resta l = { ... 3 1 0}
        System.out.println(" l.size() = " + l.size());
        System.out.println(" Cancello l_7 (contiene 2)");
        System.out.println(" l.remove(7) = " + l.remove(7));
        System.out.println(" l.size() = " + l.size());

        //Stampo senza MiniIterator: complessità in tempo alta come
        //spiegato nell'introduzione (vedi sopra), perché ogni get(i)
        //ricomincia dal primo nodo.
        System.out.println(" Stampo l usando le get(i):");
        for (int i=0; i<l.size(); i++){
            System.out.println(" " + l.get(i));
        }

        /* Per evitare una complessità in tempo alta, potrei invece consentire
        l'accesso al first di MiniLinkedList mettendolo public. Ma l'accesso

```

al first provocherebbe una "information leaking", un accesso a informazioni private della classe, che puo' produrre danni, involontari o intenzionali. Abbiamo introdotto quindi il concetto di "iterator", che è come un puntatore temporaneo, ma che sta nella heap con gli altri oggetti, non nello stack.

```
*/
```

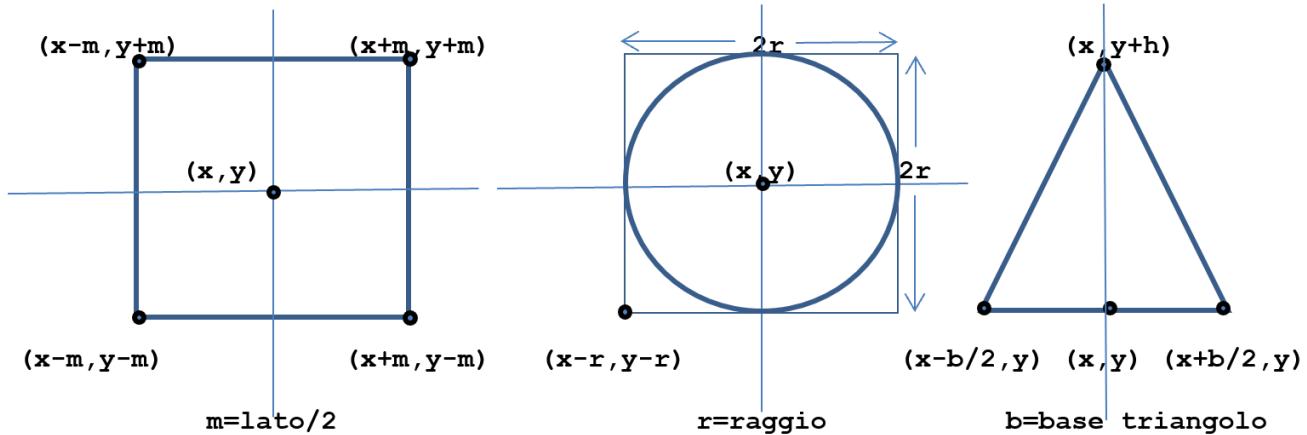
```
//Stampo tramite MiniIterator per il puntatore della stampa:  
//creo it = l.iterator() e faccio stampare utilizzando it.  
System.out.println( "Stampo l usando MiniIterator:" );  
MiniIterator it = l.iterator();  
while (it.hasNext())  
    System.out.println( " " + it.next());  
}  
  
}
```

Esercitazione 03

Array di figure

Riprendete il disegno di quadrati e cerchi 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 (interi) 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 dalle coordinate x , y del **punto medio** della base, di base b , altezza h (tutti interi, h intero col segno), e colore c . Usate queste classi per costruire una finestra Java con disegnato un array contenente quadrati, cerchi, triangoli, di differente centro (x, y) , dimensione e colore c . I costruttori delle classi avranno quindi argomenti:

Quadrato($x, y, lato, c$), **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 interfacce 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. Se cercate un colore su Wikipedia, trovate i valori di r,g,b necessari per definirlo.

Soluzione Esercitazione 03

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

//Quadrato.java
import java.awt.*;      //Abstract Window Toolkit
import javax.swing.*; //estensione di awt per interfacce 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 interfacce 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 interfacce 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);
        g.drawLine(x-b/2, y, x+b/2, y);    //base triangolo
        g.drawLine(x-b/2, y, x, y+h);      //lato sinistro
        g.drawLine(x+b/2, y, x, y+h);      //lato destro
    }
}

//Disegno.java
import java.awt.*;      //Abstract Window Toolkit (finestre grafiche)
import javax.swing.*; //estensione di awt per interfacce 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 array 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); // azzerare contenuto del frame g
    g.translate(w/2,h/2); // traslo 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);
}
```

Esercitazione 03

Estensione facoltativa dell'esercizio

Provate ad aggiungere un attributo privato **Color d** per Cerchi, Quadrati e Triangoli, per decidere il colore dell'interno della figura. Per disegnare l'interno di un poligono oppure di un ovale, e il poligono stesso, usate i seguenti comandi per oggetti della classe Graphics:

```
fillOval(int x, int y, int width, int height)
fillPolygon(int[] xPoints, int[] yPoints, int nPoints)
drawOval(int x, int y, int width, int height)
drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
```

Trovate le spiegazioni su internet. Per riempire una figura con il colore d, dovete prima usare il comando **g.setColor(d)** e dopo il comando fillOval, fillPolygon, esattamente come quando decidete il colore di una linea.

Non includiamo qui la soluzione della versione facoltativa dell'Esercitazione 03.

Lezione 15

Classi astratte di figure

Lezione 15. Parte 1. Una classe Figure per il calcolo di area e perimetro. Nella Lezione 13 abbiamo visto come **disegnare** figure diverse usando la nozione di **sottoclasse**. In questa lezione consideriamo un problema simile: definire delle classi per **calcolare area e perimetro** delle figure geometriche: cerchi, poligoni regolari, trapezi, rettangoli eccetera. Vedremo anche una versione migliorata della stessa soluzione, utilizzando le **classi astratte** di Java.

Ecco un'indicazione per 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. Questa molteplicità di tipi comporta degli svantaggi: per esempio, non possiamo raggruppare delle figure in un array 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 (come sottoclassi), e che non contenga oggetti, oltre agli oggetti costruiti nelle 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. I metodi vuoti non vengono mai usati. *Questa soluzione adatta la soluzione vista nella Lezione 13.*

Adesso vediamo una soluzione migliorata: dichiariamo una classe come astratta. Possiamo migliorare la soluzione precedente dichiarando una classe e alcuni dei suoi metodi come **astratti**. I metodi astratti rappresentano l'impegno da parte nostra di definire un metodo con quel nome nelle sottoclassi, e non si possono usare. Inoltre **non** possiamo definire un oggetto usando il costruttore di una classe astratta (possiamo invece usarlo nella definizione del costruttore di una sottoclasse). Ogni oggetto che appartiene a una classe astratta viene definito dal costruttore di una sottoclasse concreta, oppure è **null**. Il vantaggio di dichiarare una classe astratta è che Java controlla per noi: che non usiamo un metodo astratto prima di averlo sovrascritto in una sottoclasse, e che non definiamo oggetti usando il costruttore di una classe astratta.

Chiameremo **concreta** una classe non astratta.

Regole per le classi astratte. **(i)** Gli attributi non sono astratti. **(ii)** Se un **metodo è astratto la sua classe è astratta**, ma una classe astratta può avere anche metodi concreti. **(iii)** Una classe astratta può **definire costruttori**, ma questi si possono **usare solo nelle sue estensioni**. Dunque una classe astratta contiene solo null e gli oggetti costruiti nelle sottoclassi concrete. **(iv)** Perché una sottoclasse di una classe astratta sia concreta deve **sovrascrivere tutti i metodi astratti** che eredita.

Nei diagrammi UML, metodi astratti e classi astratte sono solitamente indicati scrivendone il nome in **corsivo** (noi inoltre li scriviamo in **rosso**).

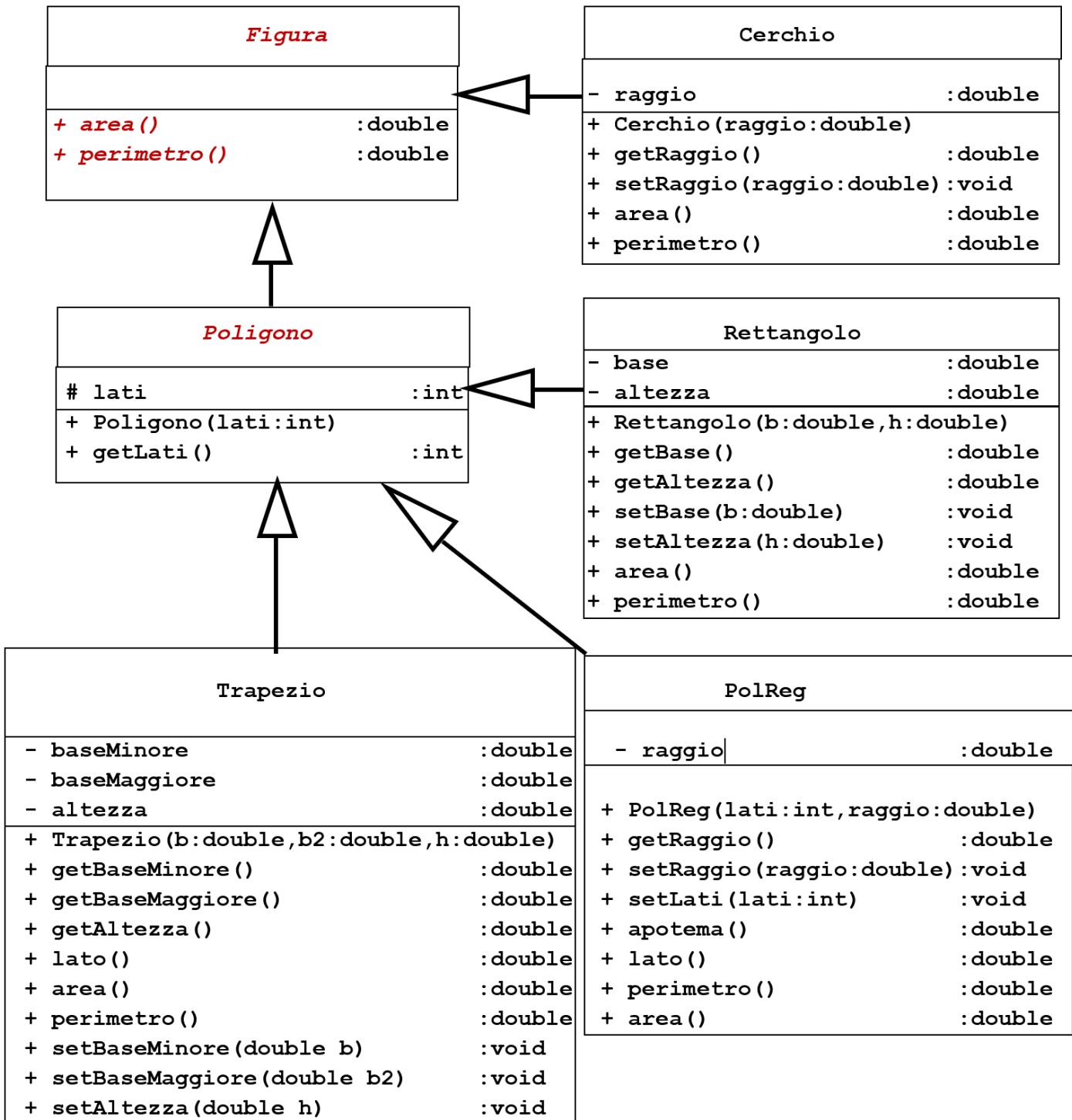
Forniamo ora il **diagramma UML di una classe Figura** di figure per cui vengono forniti metodo per il calcolo di area e perimetro (per questa classe di Figure non abbiamo metodi grafici). Questi sono i soli metodi (astratti) di Figura.

Le sottoclassi sono Cerchio, classe concreta, con l'attributo raggio e relativi get e set, e Poligono, con lati (numero dei lati) e un metodo get. Non possiamo fornire un metodo set per il numero dei lati a Poligono, altrimenti verrebbe ereditato da tutte le sottoclassi, per esempio quella dei Rettangoli, e potremmo ottenere rettangoli con più o meno di quattro lati. Poligono è una classe astratta, perché non abbiamo un metodo per il calcolo di area e perimetro che valga per ogni poligono.

Invece le sottoclassi di Poligono sono concrete: Rettangolo, Trapezio, PolReg, rispettivamente con gli attributi base, altezza, baseMinore, baseMaggiore, raggio e apotema. Per "trapezio" intendiamo "trapezio **isoscele**". Nella classe PolReg dei poligoni regolari ha senso inserire un metodo setLati(int lati) per variare il numero dei lati.

Infine, per completare la costruzione, ogni sottoclasse concreta di Figura (Cerchio e le sottoclassi concrete di Poligono) deve avere metodi concreti per il calcolo di area e perimetro.

Diagramma UML della classe astratta Figura e sottoclassi



Vediamo ora come realizzare le classi astratte del diagramma UML precedente.

```

//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 dichiarare variabili di tipo Figura:

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

    /** Però NON posso creare oggetti di tipo Figura con new Figura(),
        ma posso assegnare a P il valore null oppure un oggetto di una
        sottoclasse concreta di Figura. ATTENZIONE: il valore di default di P
        non è null. Infatti se si prova a stampare P con una println subito
        dopo la sua dichiarazione il compilatore ci dice che la variabile P
        non è stata inizializzata. */
    }

    // end class Figura

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

    public class Cerchio extends Figura{
        private double raggio; //INVARIANTE: raggio>0

        public Cerchio(double raggio){

```

```

    assert raggio >= 0;
    this.raggio = raggio;
}

public double getRaggio() {
    return raggio;
}

public void setRaggio(double raggio){
    assert raggio >= 0; //per rispettare l'invariante
    this.raggio = raggio;
}

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

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

}

// end class Cerchio

/** Poligono.java. Sottoclasse astratta di Figura:
anche se classe figlia di Figura, non ha dei metodi implementati 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; // INVARIANTE: lati>=3
    /* "protected" consente di modificare "lati" in una sottoclasse di
    Poligono (ci serve per i poligoni regolari) */

    public Poligono(int lati){
        //Non e' necessario invocare il costruttore della classe superiore
        //quando e' il costruttore di default, quindi possiamo omettere:
        //super();
        assert lati >= 3; //per mantenere l'invariante
        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 da Java.
// Posso dichiarare variabili di tipo Poligono:
public static void main(String[] args){
    Poligono P;

    //Poligono triangolo = new Poligono(3); //NON corretto:
    //triangolo non può essere creato nella classe astratta Poligono
}

/** Però NON posso creare oggetti di tipo Poligono con new
Poligono(), ma posso assegnare a P il valore null oppure un oggetto
di una sottoclasse concreta di Figura. ATTENZIONE: il valore di
default di P non è null. Infatti se si prova a stampare P con una
println subito dopo la sua dichiarazione il compilatore ci dice che
la variabile P non è stata inizializzata. */

/** 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.*/
public class PolReg extends Poligono{
    private double raggio; // invariante raggio>0

    public PolReg(int lati, double raggio){
        super(lati);
        assert raggio >= 0; // per mantenere l'invariante
        this.raggio = raggio;
    }

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

    public void setRaggio(double raggio){
        assert raggio >= 0; // per mantenere l'invariante
        this.raggio=raggio;
    }
}

```

```

}

public void setLati(int lati){ //richiede "lati" protected
    assert lati>=3; // per mantenere l'invariante
    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 posso assegnare un oggetto E della sottoclass PolReg a
una variabile di tipo Figura (classe astratta) perche' questa
sottoclass 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()); //calcolata nella classe PolReg
}
}

/** Trapezio.java (Trapezio Isoscele). Sottoclass 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;    //INV. 0 < baseMinore
    private double baseMaggiore; //INV. baseMinore <= baseMaggiore
    private double altezza;       //INV. altezza > 0

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

public void setBaseMaggiore(double baseMaggiore)
{assert baseMinore<=baseMaggiore; this.baseMaggiore=baseMaggiore; }

public void setAltezza(double altezza)
{assert altezza>0; 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));}  
}  
// end class Trapezio  
  
/** Rettangolo.java. Sottoclasse non astratta di Poligono e quindi di  
Figura. Rettangolo ha dei metodi implementati, 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; //INVARIANTE: base>0, altezza>0  
  
    public Rettangolo(double base, double altezza){  
        super(4);  
        assert base>0 && altezza>0; //per mantenere l'invariante  
        this.base=base;  
        this.altezza=altezza;  
    }  
  
    public double getBase()  
    {return base;}  
  
    public double getAltezza()  
    {return altezza;}  
  
    public void setBase(double base)  
    {assert base>0; this.base=base;}  
  
    public void setAltezza(double altezza)  
    {assert altezza>0; this.altezza=altezza;}  
  
    public double area()  
    {return base*altezza;}  
    public double perimetro()  
    {return 2*(base+altezza);}  
}  
// end class Rettangolo
```

```

//TestFigura.java
/** Classe introdotta per provare a usare la classe Figura e il
dynamic binding. La classe Figura ha solo metodi astratti 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 array di
    figure non vuoto */
    public static int maxArea(Figura[] V) {
        // esempio di dynamic binding
        // 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' Cerchio, quindi l'area di f si calcola
        con il metodo area() definito per i cerchi */
        System.out.println( "\n Area cerchio f di raggio 1 = " +
f.area());
        /** Finalmente incassiamo un vantaggio dalla classe astratta
        Figura: ora posso definire un array 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),   //Rettangolo 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, alt 3
        };
        int n = a.length;
        System.out.println

```

```
( "\n Stampo 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("\n Indice figura di massima area in a="
    + maxArea(a));
}
}
// end class TestFigura
```

Lezione 16

La classe astratta Lista

Lezione 16. Le classi astratte ricorsive. Abbiamo già visto nella Lezione 14 come rappresentare liste di oggetti usando la classe MiniLinkedList con attributo first = "l'indirizzo del primo nodo nella lista", first=null se la lista è vuota. Un'altra possibilità è rappresentare liste vuote e non vuote con oggetti **con etichetta** "lista vuota" e "lista non vuota". Rappresentiamo in Java queste etichette con la **classe** delle liste vuote e la classe delle liste non vuote. Questa rappresentazione contiene delle informazioni in più, utili a Java per scegliere quale versione di un metodo usare attraverso il dynamic binding. Otteniamo questa rappresentazione considerando le liste come una **classe astratta ricorsiva**.

Nota. Non è essenziale usare le classi astratte ricorsive nel caso delle liste: usiamo le liste solo per avere un esempio semplice. Le classi astratte ricorsive diventano importanti per problemi avanzati, quando si considerano dati con molte "forme" possibili, con ogni dato composto di un diverso numero e tipo di dati più semplici. Un esempio sono gli alberi di parsing, che vedrete nel corso di Linguaggi Formali. Ora torniamo a parlare di liste come classi astratte ricorsive.

Definiamo una classe astratta **List** delle liste ordinate senza ripetizioni di interi. List ha due sottoclassi concrete. Abbiamo la classe **Nil** che contiene (oltre a null che **NON** usiamo in questa particolare rappresentazione per rappresentare le lista vuota) gli oggetti che rappresentano le liste vuote. Abbiamo la classe **Cons** degli oggetti che rappresentano liste contenenti un elemento "elem" e una lista "next", fatta di elementi tutti più grandi di elem (oppure vuota). Dunque Cons è definito a partire da List che è definito a partire da Cons: la definizione di List e Cons è **ricorsiva**. Possiamo usare List per rappresentare **insiemi finiti di interi**. Abbiamo scelto di usare liste ordinate senza ripetizioni: questa scelta ha il vantaggio di avere una sola rappresentazione per ogni insieme di valori.

Il primo passo è scegliere quali metodi implementare (tutti dinamici e pubblici). Chiediamo di avere (i) un metodo **boolean empty()** per decidere se una lista è vuota, (ii) un metodo **int size()** per contare

gli elementi di una lista, (iii) 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. Innanzitutto il metodo (iv) **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. Quindi il metodo (v) **List append(List l)**, che costruisce una nuova lista unione della lista data e della lista **l**. Infine (vi) nelle classi Nil e Cons riscriviamo il metodo **String toString()** per liste.

In questo esercizio non definiamo metodi che modificano liste già esistenti: questa scelta ha il vantaggio di evitare il rischio di sovrascrivere dati che ancora ci servono, e lo svantaggio di usare tempo e memoria in più, per costruire le nuove liste. Sfruttiamo il più possibile le liste esistenti, come vedremo.

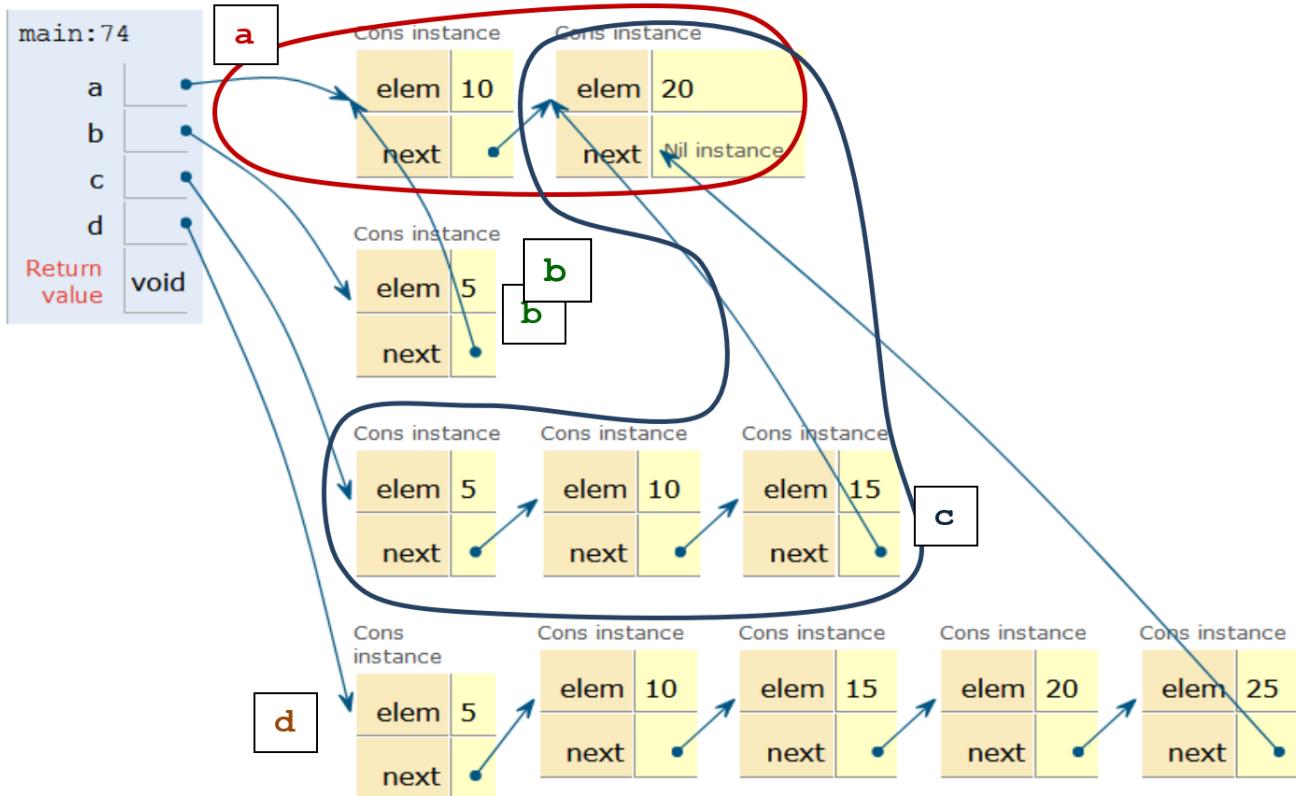
Il costruttore **Cons(int elem, List next)**, della classe Cons di liste non vuote, consente di definire liste non ordinate: per questo motivo non ne consentiamo l'uso fuori di List, della sua cartella e delle sue sottoclassi, e lo dichiariamo **protected**. Non possiamo definire Cons come **private**, perché Cons viene usato nella classe Nil e dunque fuori di Cons, anche se nella stessa cartella. La classe Nil ha il costruttore pubblico **Nil()** per la lista vuota.

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 quanto possibile le liste precedenti**. Con la particolare definizione di List che sceglieremo, otterremo il seguente risultato: 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
(ovvero condividono alcuni nodi)



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
  (non ci sono quindi ripetizioni di elementi).
  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 in Nil, Cons
    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. Nil contiene oggetti new Nil() che
rappresentano la lista vuota, e null che invece rappresenta un
elemento indefinito. In questa classe: this = oggetto istanziato =
lista vuota. */
/* NOTA. Questa classe non si puo' compilare prima di aver
sovrascritto tutti i metodi astratti di List, ne' senza la 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 un elemento della classe Nil, la lista vuota. Evitando di scrivere new Nil() riutilizzo la precedente lista vuota, ovvero quella su cui ho chiamato il metodo. */

/** 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. */
} // end class Nil

//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 (con null elemento indefinito). Nella definizione ricorsiva di un metodo di Cons usiamo metodi di List, che a seconda del tipo esatto dell'oggetto List indicano un metodo di Cons o di Nil. */

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 in modo che non sia accessibile da un programma esterno, perche' consente di costruire liste non ordinate, mentre vogliamo impedire a chi importa una cartella con la classe List di farlo. Usando protected possiamo usare Cons in Nil, che si trova nella stessa cartella. */
    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);}
// 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 elem aggiungo x davanti a tutti:
    if (x < elem)
        return new Cons(x, this);
    //Se x uguale al primo elem 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;//questo test non e' strettamente
    //necessario, ma rende la computazione piu' efficiente
    else return next.append(l).insert(elem);
}

```

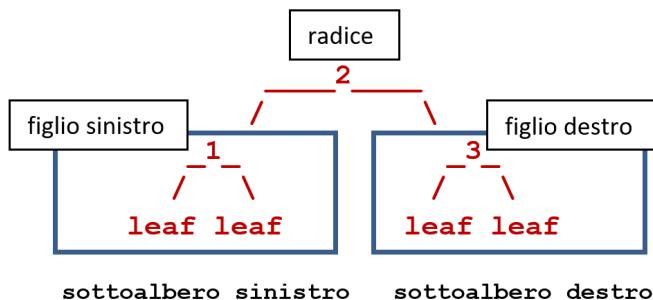

Lezione 17

La classe astratta ricorsiva degli alberi di ricerca

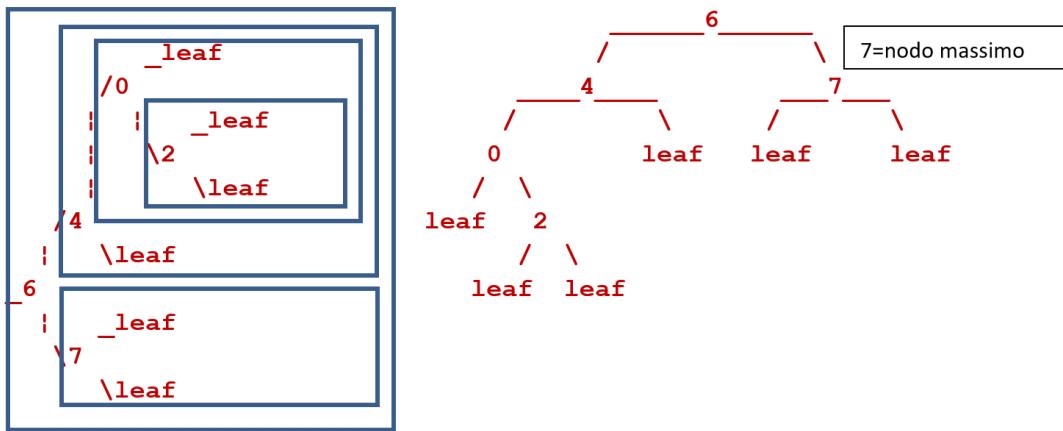
Lezione 17. La definizione ricorsiva di albero binario. Vediamo ora un altro esempio di classe astratta e ricorsiva: la classe degli **alberi di ricerca**. Si tratta di una classe importante per definire una struttura dati che sia dinamica (non ha una dimensione prefissata, cresce o diminuisce a seconda delle esigenze) e che consenta nel caso medio un accesso veloce (in tempo logaritmico) ai dati. Come esempio vedremo alberi di ricerca di interi, che rappresentano insiemi finiti di interi, ma con poco più lavoro potremmo definire una classe generica, di alberi di ricerca su un tipo di dati T qualsiasi.

Iniziamo con la definizione ricorsiva degli **alberi binari di interi**. Gli alberi binari di interi sono **alberi vuoti**, rappresentati **con oggetti di tipo 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**. Il percorso dalla radice a un nodo è detto un **ramo** dell'albero.

Un albero binario di interi è **di ricerca** se la radice è maggiore di ogni nodo 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**. Il termine "di ricerca" si riferisce al fatto che per questi alberi la ricerca di un elemento nell'albero è particolarmente efficiente: il tempo richiesto è in media proporzionale al logaritmo del numero dei nodi. Vedrete questi argomenti nel corso di Algoritmi. 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 disegno** 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.



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 classe astratta e ricorsiva Tree ha due sottoclassi concrete, **Leaf** (alberi vuoti) e **Branch** (alberi non vuoti), gli elementi di Branch possono contenere elementi di Leaf e altri elementi di Branch.

Accenniamo anche che ci sono esempi di classi astratte ricorsive con **molte sottoclassi concrete**: per esempio la classe delle istruzioni di un linguaggio di programmazione si può rappresentare come una classe astratta ricorsiva con molte sottoclassi concrete, la classe degli **If**, la classe dei **While**, dei **For**, delle **assegnazioni** e così via. Tuttavia, questi esempi fanno parte di corsi successivi, come Linguaggi Formali, e non li vedremo in questo corso.

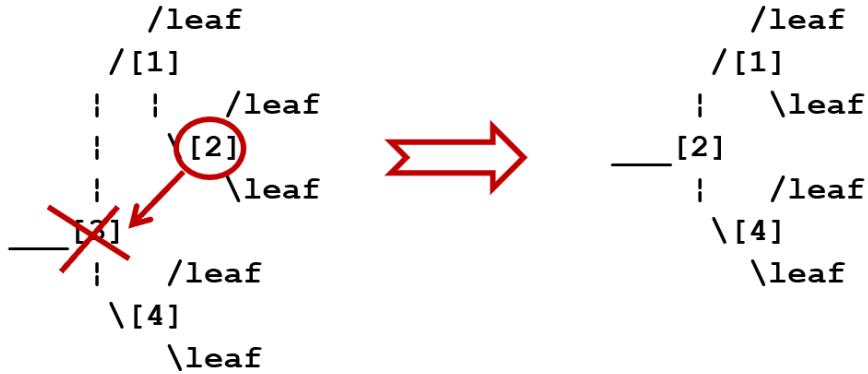
Ritorniamo ora alla classe Tree. Gli elementi di Tree hanno la caratteristica di essere **dinamici** (cambiano di dimensione a seconda della necessità). Come abbiamo anticipato, inoltre, consentono di effettuare ricerca, inserimento e cancellazione di un nodo in **tempo**

proporzionale al **logaritmo del numero dei nodi** (quindi velocemente). Questo, purché siano mantenuti **bilanciati** (cioè con parte sinistra e destra approssivamente uguali). Nel corso di ProgII non ci occupiamo di questo secondo aspetto, che sarà trattato nel corso di Algoritmi.

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

Infine aggiungiamo **(vi)** un metodo **String toString()** per "disegnare" un albero con soli caratteri ASCII, metodo che non spieghiamo.

Nota 1. Come rimuovere 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 dell'albero e entrambi i suoi sottoalberi non sono vuoti**. Prendiamo l'albero con radice 3, figli sinistro e destri 1 e 4, e con 2 figlio destro di 1. Ecco un disegno fatto con soli caratteri ascii.



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 riconnettere i due alberi, con una radice **scelta in modo tale da ottenere un albero di ricerca**. A tal fine, rimpiazziamo 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 dell'albero di partenza, questa è una definizione corretta di un algoritmo ricorsivo per rimuovere un nodo da un albero.

Nota 2. Cosa succede quando modifichiamo un albero. In questa implementazione, quando inseriamo o cancelliamo elementi **scegliamo di modificare l'albero originale**. Questo significa che se scriviamo per esempio `t.remove(x)`, l'indirizzo dell'albero `t` **può cambiare**, e adesso la variabile `t` può contenere un indirizzo errato. Per ovviare a questo scriveremo `t=t.remove(x)`, per rimuovere un elemento `x` e contemporaneamente aggiornare l'indirizzo di `t`. *Se ce ne dimentichiamo, generiamo errori a catena nel nostro programma.*

```
// 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 contenuto precedente 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 String toStringAux
    (String prefix, String root, String left, String right);
//Metodo che gestisce la parte NON pubblica della stampa.2

public String toString()
    {return toStringAux("", "___", "    ", "    ");}

/* Trascrizione albero --> stringa. Ogni albero viene trascritto in
stringa dall'alto verso il basso, un elemento per riga, con i
sottoalberi disegnati piu' a destra dell'albero di cui fan parte. Il
risultato e' un disegno bidimensionale fatto con soli caratteri
ascii. */
}

// end class Tree

//Leaf.java: i suoi oggetti (diversi da null) rappresentano alberi
//vuoti

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

```

² Non e' essenziale capirlo. Solo per curiosità: "prefix" è la parte iniziale di ogni riga del disegno (la parte sinistra di ogni riga). A destra di "prefix" in ogni riga aggiungiamo: (i) "left" in ogni riga che rappresenta il sotto-albero sinistro (sono le righe in alto), (ii) "root" una volta sola nella riga che rappresenta la radice (e' la riga in mezzo), (iii) "right" in ogni riga che rappresenta il sotto-albero sinistro (sono le righe in basso).

```

prefix left ... | (disegno sotto-albero sinistro) |
prefix left ... | (disegno sotto-albero sinistro) |
prefix root ... radice
prefix right... | (disegno sotto-albero destro)  |
prefix right... | (disegno sotto-albero destro)  |

```

Da questa definizione si possono dedurre tutte le clausole della definizione ricorsiva di toStringAux. Noi non lo facciamo.

```

public class Leaf extends Tree {
    public Leaf() { }

    /** Il costruttore new Leaf() non assegna nulla e si puo' anche
     lasciare implicito. Un albero viene inizializzato da new Leaf() e poi
     esteso un elemento alla volta. Qui this = oggetto istanziato = albero
     vuoto (sempre). */

    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
    //qui this è un oggetto di tipo Leaf: lo riuso, quindi non creo
    //alberi vuoti nuovi.
    //Nella classe Leaf abbiamo sempre: this = albero vuoto

    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 String toStringAux
    (String prefix, String root, String left, String right)
        {return prefix + root + "leaf";}
}

// end class Leaf

    /**
     * Sottoclasse 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 va 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 del sottoalbero
            // sinistro (massimo dei minimi)
            // e, per evitare ripetizioni, eliminiamo
            // il massimo dal sottoalbero sinistro:
            left = left.remove(elem); //aggiorno 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,
// non e' essenziale.

protected String toStringAux
(String prefix, String root, String left, String right){
    return this.left.toStringAux(prefix+left, " /", " ", " | ")
        + "\n" + prefix + root + "[" + elem + "]" + "\n" +

```

```

        this.right.toStringAux(prefix+right, "    \\\", \" |\", \" ");
    }
}

// end class Branch

//TestTree.java. Per sperimentare la classe Tree degli alberi di
//ricerca
/* Per stampare un albero t, trascrivo t nella stringa t.toString(),
con toString() ridefinito. Il comando e': System.out.println(t),
abbreviazione di System.out.println(t.toString()) */
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(System.currentTimeMillis());
        //r e` un generatore di numeri pseudo-casuali definito a partire
        //dall'orologio di sistema.

        // 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 \n\n" + t + "\n\n t.max() = " + t.max());

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

        //Creo un albero u inserendo sempre elementi piu' piccoli
        //quindi sempre nella parte sinistra dell'albero
    }
}

```

```

Tree v = new Leaf();
for (int i = n-1; i >=0; i--) v = v.insert(i);
System.out.println( "\n Stampa albero v di " + n + " elementi,
elementi, tutti figli sinistri \n\n" + v);

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

/*
Stampa albero casuale t di al piu' 10 elementi tra 0 e 100

      /leaf
      /[8]
      |  \leaf
      / [12]
      |  |    /leaf
      |  |    / [28]
      |  |    |  \leaf
      |  |    / [35]
      |  |    |  \leaf
      |  \ [50]
      |    |  /leaf
      |    \ [56]
      |      \leaf
_____[77]
      |    /leaf
      |    / [80]
      |    |  \leaf
      |    / [90]
      |    |  \leaf
      \ [92]
        \leaf

t.max() = 92
*/

```

Lezione 18

Generici vincolati, interfacce e alberi di ricerca

Lezione 18. In questa lezione introduciamo i *generici vincolati*, cioè le variabili di classe T con restrizioni o vincoli. Come **esempio di vincolo**, chiediamo che la classe T abbia una **relazione di ordine**, e usiamo T per definire una classe astratta degli **alberi di ricerca su una generica classe T**.

Interfacce Java. Per spiegare la nozione di vincolo su una classe generica T è necessario prima conoscere la nozione di interfaccia in Java. Una interfaccia è **una classe astratta**: si presenta come una lista di signature di metodi dinamici, quindi metodi astratti, **senza nessuna implementazione**. Un'interfaccia può contenere metodi statici implementati. Una classe astratta può avere degli attributi e dei metodi concreti, una interfaccia no. Una interfaccia viene definita usando la parola chiave "interface". 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 astratti. Una classe Java può estendere al più una classe, ma **può implementare un numero qualunque di interfacce**, scrivendo **C implements I, J, ...**. Notiamo che nel caso delle classi (astratte e concrete), se per esempio abbiamo due classi A, B che forniscono due versioni concrete diverse dello stesso metodo m, se la classe C estendesse A, B si avrebbe un conflitto su quale versione di m ereditare. Per questa ragione, Java non consente l'ereditarietà multipla tra una classe e due o più classi. Questo conflitto non succede aggiungendo un numero qualsiasi di interfacce: se C implementa due interfacce I, J e m compare sia in I che in J, allora I, J non forniscono versioni concrete del metodo m, e quindi non si crea un conflitto.

Un esempio di deduzione a partire dalle definizioni date. Supponiamo di avere B che estende A che implementa l'interfaccia I: qual è la relazione tra B ed I? A sovrascrive tutti i metodi di I, dunque lo stesso vale per B che estende A. Concludiamo: **B implementa I**.

L'interfaccia Comparable<T>. Per definire gli alberi di ricerca su una generica classe T, abbiamo bisogno dell'interfaccia predefinita **Comparable<T>**. Questa interfaccia ha unico metodo astratto **int compareTo(T y)**. Questo metodo viene interpretato come un confronto tra

due elementi di T: se y è in T, allora `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 una classe C implementa Comparable<C>, siamo tenuti a fornire in C una implementazione per `int compareTo(C y)`. Così facendo scegliamo una nozione di eguaglianza e una nozione di ordine per C.

Come esempio di `compareTo`: le classi predefinite **`Integer`, `Double`, `String`** implementano Comparable<T> rispettivamente per T=Integer, Double, String. Se T=Integer, Double, allora `x.compareTo(y)` è l'ordine tra numeri. Se T=String, allora `x.compareTo(y)` è l'ordine lessicografico (l'ordine del vocabolario).

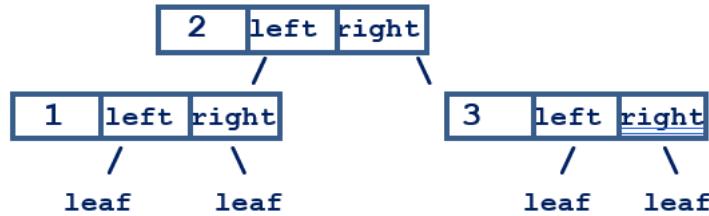
Generici Vincolati. Possiamo definire una classe astratta Tree<T> di alberi di ricerca su T aggiungendo un vincolo alla classe generica T che utilizziamo: possiamo scrivere **`Tree<T extends I>`**, dove I è una classe astratta oppure una interfaccia (bizzarramente, qui si scrive "T extends I" e non "T implements I"). In questo caso quando descriviamo Tree<T> abbiamo a disposizione tutti i metodi di I: per esempio il metodo `x.compareTo(y)` se I=Comparable<T>. 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 (se I è una classe) oppure implementa I (se I è una interfaccia), 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 l'interfaccia I = Comparable<T>, ovvero che T abbia una relazione che interpretiamo come una **relazione di ordine**. 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 di albero di ricerca. Ecco una **rappresentazione semplificata** di un albero di ricerca t={1,2,3} in Java. Il tipo di t è Tree<T>, ottenuto istanziando su T=Integer. L'albero t ha radice 2 e figli 1 e 3. La radice di t contiene l'elemento 2, e l'indirizzo dei sottoalberi t.left e t.right (sinistri e destri) dell'albero. Nel

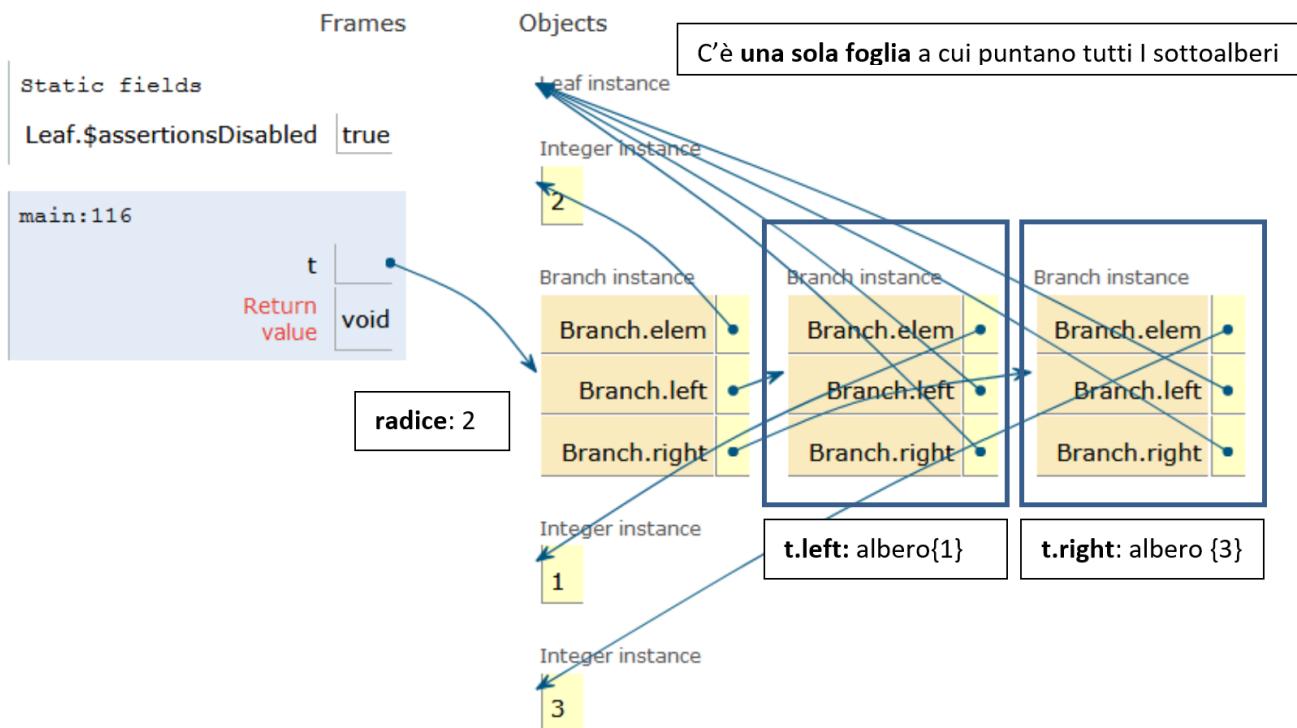
disegno poniamo $t.left$ a sinistra e $t.right$ a destra, consideriamo le foglie di t distinte tra loro e gli interi 1,2,3 parte dell'albero.

L'albero di ricerca $t=\{1,2,3\}$



Quando scriviamo un programma, di solito **ci basta** la rappresentazione semplificata vista qui sopra. Accenniamo che la rappresentazione precisa di t nella memoria di un programma Java è più complessa: gli elementi di Integer non sono parte dell'albero ma sono delle "box" esterne di contenuto 1,2,3, inoltre è possibile che le foglie di t vengano identificate tra loro per risparmiare spazio. In questo caso dai nodi contenenti 1,3 partono quattro frecce che arrivano allo stesso elemento di Leaf.

L'albero di ricerca $t=\{1,2,3\}$: rappresentazione con Stack + Heap



Proviamo ora ad applicare tutte le idee che abbiamo appena visto per realizzare la classe degli alberi di ricerca sopra una generica classe T, con una relazione di ordine rappresentata dal metodo compareTo dell'interfaccia Compare<T>. Come abbiamo detto, questa richiesta si traduce nel vincolo: **T extends Compare<T>**. La definizione di **Tree<T extends Comparable<T>>** sarà molto simile alla definizione della classe Tree degli alberi di ricerca su interi (Lezione 17). Come test per il codice scritto, sceglieremo T = la classe **Contatto** dei contatti, classe in cui siamo tenuti a implementare l'interfaccia Comparable<T>. Otteneremo così: la classe **Tree<Contatto>** degli alberi di ricerca sui contatti.

```
// Tree.Java
// Alberi di ricerca su una classe T
// vincolata a estendere l'interfaccia Comparable<T>.
// Questo vuol dire che ogni classe che instanzia T
// dovrà implementare Comparable<T> e quindi avere un metodo
// compareTo().

public abstract class Tree<T extends Comparable<T>>{
    //alberi di ricerca su T con relazione di ordine x.compareTo(y)
    public abstract boolean empty();
    public abstract boolean contains(T x);
    public abstract T max();

    //insert e remove restituiscono l'indirizzo dell'albero modificato
    public abstract Tree<T> insert(T x);
    public abstract Tree<T> remove(T x);

    protected abstract String toStringAux
        (String prefix, String root, String left, String right);
    //Metodo che gestisce la parte NON pubblica della stampa.
    //Non forniamo spiegazioni sul suo funzionamento, non e' essenziale

    public String toString()
        {return toStringAux("", "___", "    ", "    ");}

    /* Trascrizione albero --> stringa. Ogni albero viene trascritto in
    stringa dall'alto verso il basso, con i sottoalberi disegnati piu' a
    destra dell'albero di cui fan parte */
}
```

```

}

//end class Tree


//Leaf.Java
//ALBERI VUOTI con unico elemento (diverso da null): "leaf"
//non definisco nessun costruttore: di default ho new Leaf()

public class Leaf<T extends Comparable<T>> extends Tree<T>{
    //contiene albero vuoto e null, qui this = albero vuoto sempre
    public boolean empty(){return true;}
    public boolean contains(T x){return false;}
    public T max(){assert false; return null;}

    //insert e remove restituiscono l'indirizzo dell'albero modificato
    public Tree<T> insert(T x){return new Branch<T>(x, this, this);}
    public Tree<T> remove(T x){return this;}

    protected String toStringAux
        (String prefix, String root, String left, String right)
        {return prefix + root + "leaf";}
}

// end class Leaf


//Branch.java
//alberi di ricerca generici e non vuoti
public class Branch<T extends Comparable<T>> extends Tree<T>{
    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;}

    /* Per fare i confronti usiamo il metodo compareTo(), in modo da avere
       un confronto generale per tutti i T che implementano l'interfaccia
       Comparable<T> e che quindi hanno un metodo compareTo(). */
}

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();
}

//insert e remove restituiscono l'indirizzo dell'albero modificato
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;
    // restituisco l'indirizzo dell'albero modificato
}

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(); //sost. elem con il max a sx
        left.remove(elem); //per evitare ripetizioni
        return this;
        // restituisco l'albero modificato
    }
}

//Metodo che gestisce la parte NON pubblica della stampa.
//Non forniamo spiegazioni sul suo funzionamento, non e'
//essenziale.
protected String toStringAux
(String prefix, String root, String left, String right){
    return this.left.toStringAux(prefix+left, " /", " ", " |")
        + "\n" + prefix + root + "[" + elem + "]" + "\n" +
    this.right.toStringAux(prefix+right, " \\", " |", " ");
}
}
// end class Branch

//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, String email)
        {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 email)
        {this.email=email;}
    public String toString()
        {return nome + "<" + email + ">";}

    //Implemento un metodo compareTo() nella classe Contatto
    public int compareTo(Contatto x)
        {return this.nome.compareTo(x.nome);}
}
// end class Contatto

```

```

//TestTree.java. Instanzio Tree su diverse classi che implementano
//l'interfaccia Comparable<T>

import java.util.*; //per la classe Random

//Provo l'implementazione degli alberi binari di ricerca

public class TestTree {

    public static void Title(String s){ //Stampa di un titolo
        System.out.println( "-----"
            + "\n" + s + "\n" +
            "-----");}

    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<>(); //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");
        System.out.println(t + "\n\n t.max() = " + t.max());

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

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

```

```
//Provo il metodo di cancellazione per un albero di ogg. Contatto
Tree<Contatto> w = new Leaf<>();
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}");
System.out.println(w);
Title("w senza il contatto c"); w.remove(c);
System.out.println(w);
}
}
// end class TestTree
```

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()** (che importiamo dalla libreria Java con **import java.util.*;**) per ordinare un array di bottiglie, e per la ricerca binaria in un array ordinato di bottiglie. Questi metodi possono essere applicati a array su una classe T qualunque, ma hanno un vincolo: **T deve implementare Comparable<T>**.

Ricordiamo che il metodo **int compareTo(Bottiglia b)** deve restituire 0 se l'oggetto this è uguale all'oggetto b passato come parametro, un valore > 0 se l'oggetto this è più 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. Nulla ci vieterebbe di fare una scelta diversa, confrontando le bottiglie in base alla capacità.

Un dettaglio: nella Parte 1 della lezione utilizziamo per la prima volta il costrutto **foreach** su array, definito da **for(C c:v){...c...}** per v **array** 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
//aggiungendo un confronto tra bottiglie basato sul solo livello.
public class Bottiglia implements Comparable<Bottiglia> {
    private int capacita, livello; // 0 <= livello <= capacita!=0

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

```

// 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) && (capacita!=0);
    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) && (capacita!=0);
    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) && (capacita!=0);
}

public String toString(){
    return livello + "/" + capacita;
}

/** Siccome Bottiglia implements Comparable<Bottiglia>, dobbiamo
     fornire un metodo compareTo(): */
public int compareTo(Bottiglia b)
    {return this.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
}

// end class Bottiglia

// ComparaBottiglie.java - Classe di test
import java.util.*; //Per la classe Arrays

// COSA ABBIAMO FATTO: nella classe Bottiglia abbiamo aggiunto
// il metodo compareTo(), per confrontare due bottiglie
// COSA OTTENIAMO: possiamo usare i metodi statici di libreria
// Arrays.sort() e Arrays.binarySearch() per ordinare un array di
// bottiglie e per la ricerca binaria in un array 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
        System.out.println("\n b1=" + b1 + "\n b2=" + b2 + "\n b3=" + b3 +
"\n b4=" + b4);

        //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 array 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 array posso usare il "foreach"
// (La classe Arrays implementa l'interfaccia Iterable<T>):
for (Bottiglia b : bottiglie)
    System.out.println(b);

//Posso eseguire la ricerca binaria della posizione di una bottiglia
//in questo array ordinato perché 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));
}
}

//end class ComparaBottiglie

```

Lezione 19. Parte 2. Le interfacce Comparable<T>, Iterable<E> e Iterator<E>. Una classe T che implementa due interfacce. 65 minuti.

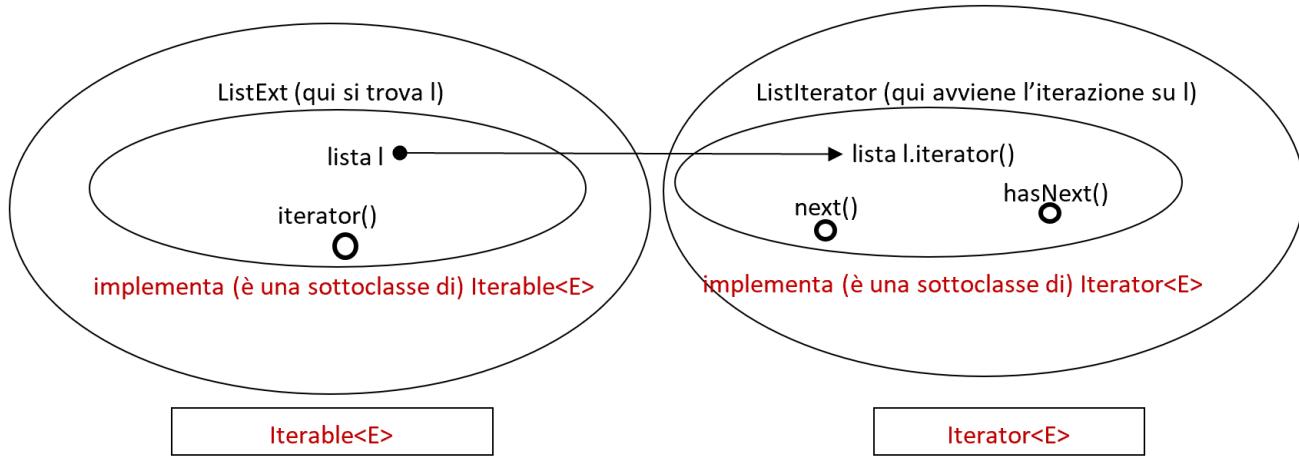
L'interfaccia Iterable<E>. Spieghiamo l'uso di Iterable<E>. Supponiamo di voler scrivere un for che passi attraverso una lista l di elementi di tipo E=Integer, ma **senza rendere pubblici gli indirizzi dei nodi all'interno di l**. Lo scopo è impedire di alterare dall'esterno la lista l. Riprendiamo la soluzione vista nella **Lezione 14**, questa volta usando le **interfacce Iterable<E> e Iterator<E>**: il vantaggio è che questa volta Java, attraverso le interfacce, può controllare se la costruzione che facciamo è corretta.

Usiamo come esempio una classe **ListExt** di liste di oggetti di tipo Integer, fornite di un attributo "size". Non vogliamo consentire di modificare il numero degli elementi di una lista dall'esterno, perché aggiungere o togliere elementi produrrebbe una inconsistenza con il valore di size.

Per impedirlo, definiamo in ListExt una implementazione per l'interfaccia **Iterable<Integer>**. L'implementazione richiede un metodo **public ListIterator iterator()** che ri-etichetta una lista l di ListExt (dunque di Iterable<Integer>) come un oggetto di **ListIterator**. Qui avviene (in modo sicuro, ovvero senza esporre i puntatori all'esterno, causandone leaking) l'iterazione su l.

Per eseguire l'iterazione in modo sicuro, ListIterator implementa (dunque è una sottoclasse di) una **seconda interfaccia, Iterator<E>**, dove E=Integer. L'interfaccia Iterator<Integer> contiene il minimo necessario per eseguire l'iterazione: il metodo **boolean hasNext()**, per sapere quando la traversata di una lista l è finita, e il metodo **Integer next()**, per conoscere il valore dell'elemento e:E contenuto nel nodo corrente di l, e per spostarsi al nodo successivo.

Il prossimo disegno riassume tutta la situazione: abbreviamo sempre **E=Integer**. ListExt appare incluso in Iterable<E> e ListIterator appare incluso in Iterator<E>.



Implementati Iterable<Integer> in ListExt e Iterator<Integer> in ListIterator, possiamo usare in ListExt usare il costrutto *foreach*: **for (Integer e:l){...e...}**, dove l ha tipo ListExt. Un foreach rietichetta l in ListIterator, quindi esegue {...e...} un volta per ogni elemento e della lista l, nell'ordine, usando i metodi hasNext(), next() di ListIterator. In tutto il processo, gli indirizzi dei nodi di l non vengono resi pubblici.

Ci sono anche altri vantaggi nell'usare l'interfaccia Iterable<Integer>. Questa interfaccia ci consente di utilizzare al

posto dell'iterazione sequenziale l'iterazione **parallela**, molto più veloce, usando il metodo **splitIterator()** e una seconda interfaccia **SplitIterator<E>**. Non possiamo approfondire questo aspetto in questo corso.

Abbiamo bisogno di importare pacchetto java.util dove risiedono le interfacce **Iterator<E>** e **Iterable<E>**. Ricordiamo anche (ne abbiamo già parlato) 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 e Java fa il passaggio da **int** e **Integer** e viceversa in modo automatico (una operazione detta **autoboxing**).

Un esempio di una classe che implementa due interfacce. Per aggiungere un esempio di classe che implementa due interfacce, scegliamo di implementare in **ListExt** una seconda interfaccia, **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 c'è un nodo successivo)
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 interfacce Iterable<T>, Comparable<T>

// Dichiariamo che ListExt implementa le interfacce Iterable<Integer>
// (1) e Comparable <ListExt> (2).
// Questo consentira' da un lato di usare il costrutto
// iterativo foreach di Java per iterare sugli elementi di una
// struttura (1) e dall'altro di confrontare istanze della classe
// ListExt secondo l'ordine lessicografico (2).

public class ListExt implements Iterable<Integer>,
Comparable<ListExt> {
    private Node first;
    private int size;
    //INVARIANTE: size = lunghezza lista nodi che parte da first

    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 Iterable<Integer>. Definiamo iterator()
// un metodo che copia un oggetto di ListExt in un
// oggetto di ListIterator, una classe che implementa
// Iterator<Integer> (si veda la classe ListIterator sotto).
public Iterator<Integer> iterator(){
    return new ListIterator(first); //risultato in Iterator<Integer>
}

// Implementazione di Comparable<ListExt>, fornendo
// compareTo(): confrontiamo due liste, this e lista,
// 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
}
}// end class ListExt

//ListIterator.java Questa classe deve implementare Iterator<Integer>
import java.util.*;

// un oggetto di ListIterator e' un semplice indirizzo di un nodo
public class ListIterator implements Iterator<Integer>{
    private Node next; //All'inizio = inizio lista
    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.
    Notiamo che il valore originario di next viene perso, quindi la
    visita si fa una volta sola, per una seconda visita bisogna
    ricalcolare l.iterator() */

    public boolean hasNext(){return next != null;}

```

```

public Integer next() {
    int x = next.getNext(); //contenuto del prossimo nodo
    next = next.getNext(); //indirizzo del nodo dopo ancora
    return x;
    //x viene automaticamente trasformato da int a Integer
    // (via auto-boxing)
}
} // end class ListIterator

//TestList.java. Sperimentiamo foreach e compareTo
//Se usiamo foreach non dobbiamo esplicitamente usare iterator()
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));
    }
} //end class TestList

```

Lezione 20

Eccezioni controllate e non controllate

Eccezioni. Una eccezione in Java è un oggetto della classe ***Exception*** e viene usato per segnalare una situazione che il programma non sa gestire. Ci sono classi di eccezioni che descrivono **errori interni** al programma scoperti a runtime. Avete di sicuro già visto comparire queste eccezioni come messaggi di errore quando un programma non conclude la sua esecuzione in modo corretto. Sono per esempio:

- ***ArithmetricException*** (divisione per zero, radice di un numero negativo),
- ***IllegalArgumentException*** (un metodo riceve valori in input non corretti per quel metodo, per es., cerchiamo una **parola vuota** in un dizionario italiano),
- ***IllegalStateException*** (lo stato del programma non consente l'uso di un dato metodo, per es., aggiungiamo un elemento a uno stack **già pieno**),
- ***ArrayIndexOutOfBoundsException*** (tentativo di accedere a un elemento **inesistente** di un array),
- ***NullPointerException*** (tentativo di accedere a **un attributo o metodo** dell'oggetto indefinito null).

Vediamo qualche esempio di questo primo gruppo di eccezioni e di altre. Definiamo dei metodi che vengono accettati dal compilatore ma generano errori a runtime.

```
//EsempiEccezioni.java
/* Per diverse eccezioni che descrivono errori interni a un programma
scriviamo un metodo che solleva quella eccezione */
public class EsempiEccezioni {

    public static void test_ArithmetricException()
    {System.out.println(1 / 0); } //divisione per 0

    public static void test_ClassCastException()
    {Object obj = "ciao"; Float f = (Float) obj;}
```

```

/* Downcast errato: il tipo esatto di obj e' String, e non e' possibile
convertire String a Float */

public static void test_NumberFormatException ()
{Integer.parseInt("ciao");} /* parseInt trasforma una stringa che
rappresenta un intero in quell'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 come toString 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();
}

}

```

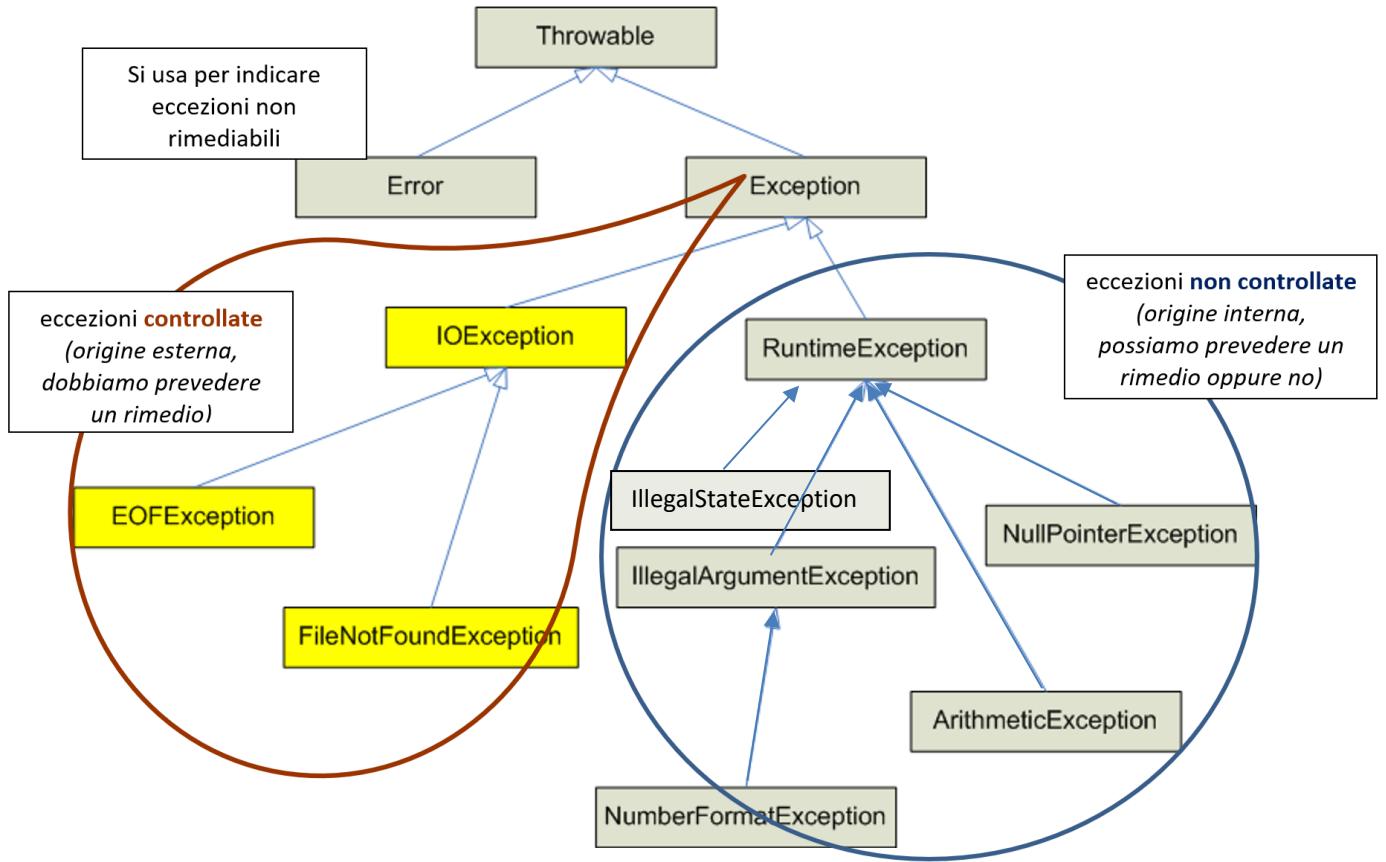
Ci sono altri tipi di eccezioni, tipicamente causate da **errori esterni** al programma. Per esempio: **FileNotFoundException**, quando chiedo di aprire un file inesistente, oppure **IOException**, quando ho errori durante la lettura/scrittura da un file o dalla rete. Per avere le classi di eccezioni IO devo scrivere **import java.io.*;**

Infine, ci sono gli errori, che sono eccezioni particolarmente gravi. Per esempio, un comando **assert cond:"messaggio";** solleva un errore di tipo **AssertionError** quando la condizione cond è falsa, terminando così il programma, e invia un messaggio di errore. Abbiamo già visto come usare un assert per segnalare errori.

Eccezioni non controllate e controllate. Si fa una prima distinzione tra le eccezioni causate da errori interni al programma, che vengono dette **eccezioni non controllate**, e quelle prodotte da cause esterne, che vengono dette **eccezioni controllate**. Le eccezioni non controllate

vengono chiamate così perché **è responsabilità del programmatore decidere se gestirle** quando accadono, prevedendo un rimedio, oppure no. Gestire una eccezione è complicato, come vedremo, ma anche non gestirle può essere molto costoso. Il programmatore, se non vuole gestire le eccezioni, può limitarsi a cercare tutti gli errori che impediscono al programma di non terminare correttamente, evitando la loro comparsa, ma senza prevedere un piano di emergenza quando accadono. Invece, il programmatore è tenuto a trattare **sempre** le eccezioni controllate che ha inserito con una parte di programma apposita, perché queste ultime sono causate da eventi che non dipendono dal suo codice, ma da fattori esterni che non si possono prevedere nel programma stesso.

Vediamo le relazioni di sottoclasse tra questi due tipi di eccezioni. La classe Eccezione contiene sottoclassi di eccezioni controllate ("checked"), come IOException (input/output Exception), e sottoclassi più piccole, come EOFException, FileNotFoundException. Sempre la classe Eccezione contiene classi di eccezioni non controllate ("unchecked"), come RunTimeException, e sottoclassi più piccole, come NullPointerException. NullPointerException è forse l'eccezione non controllata più comune. La classe Errore è disgiunta da Eccezione e considerata non controllata. Entrambe le classi sono incluse in Throwable, la classe di tutti gli errori e di tutte le eccezioni che possono essere sollevate.



Cattura di eccezioni. Non sempre è desiderabile terminare un programma in presenza di una eccezione: per esempio un programma non deve terminare se cerca di aprire un file e non lo trova, questa situazione è comune e quindi deve essere prevista. Per evitare di terminare un programma, le eccezioni possono essere **catturate**: basta scrivere il codice nella forma **try{...} catch(TipoEccezione x){...}**. Questo codice inizia eseguendo il corpo di **try**. Poi, se viene sollevata una eccezione t di tipo TipoEccezioni, anziché terminare il programma esegue il corpo di **catch**, con x assegnata ad t. Si consiglia di usare la classe **Error** per situazioni in cui non esiste rimedio, e di **evitare la cattura** di un errore (cattura che però è possibile in Java).

Possiamo scegliere quale tipo di eccezione sollevare scrivendo in un metodo **void m()**

throw new TipoEccezione(...);

Posso avvisare Java che **m()** può sollevare una eccezione di tipo TipoEccezione aggiungendo alla segnatura del metodo, scrivendo: **void m() throws TipoEccezione**. In tal caso **m()** può essere usato solo all'interno di **un codice che cattura** l'eccezione **TipoEccezione**.

Supponiamo che m() contenga `throw new TipoEccezione(...)`. Allora le regole per throws sono:

1. Nel caso TipoEccezione sia una eccezione non controllata, non sono obbligato a scrivere throws nella segnatura di m()
2. Nel caso TipoEccezione sia una eccezione controllata, sono obbligato a scrivere "throws TipoEccezione" nella segnatura di m()

Vediamo un primo esempio di catch/throw/throws:

```
import java.io.*; //per le IOException
public class TestError{
    public static void m()
    //Error e' non controllato, non sono obbligato scrivere throws.
    //Ma potrei farlo: se lo facessi, dovrei mettere le sue chiamate
    //in un costrutto "try-catch"
    {throw new Error( "error" );}
    //viene chiamato il costruttore della classe Error, che prende
    //un parametro String: ricordatevi, le eccezioni sono oggetti!

    public static void m2() throws IOException
        //IOException e' controllata, devo scrivere throws
        //e le chiamate di m2() devono stare in un try-catch
        {throw new IOException( "io exception" );}

    public static void m3()
        //RuntimeException e' non controllata, non devo scrivere throws
        //ma posso farlo: se lo faccio, devo mettere le sue chiamate in
        //un try-catch
        {throw new RuntimeException( "runtime exception" );}

    public static void main(String[] args){
        try {m();m2();m3();}
        catch (Throwable e) //catturo tutti gli errori o eccezioni
        {System.out.println("Captured: " + e);}
        //finito il body del catch, non si prosegue nel try,
        //ma si prosegue con il programma:
        System.out.println("Sono fuori dal try-catch!");
    }
}
```

```
/* NOTA.Un altro esempio.  
Se invece scrivo:  
  
public static void main(String[] args){  
    m2();  
    try {m(); m3();}  
    catch (Throwable e) //catturo tutti gli errori o eccezioni  
        {System.out.println("Captured: " + e);}  
    System.out.println("Sono fuori dal try-catch!");  
}
```

Il compilatore dice:

```
TestError.java:21: error: unreported exception IOException;  
must be caught or declared to be thrown  
m2();  
^
```

perche' nella firma di m2() c'è la clausola 'throws' che obbliga la gestione (o a un nuovo lancio) dell'eccezione sollevata. Siccome quell'eccezione è "controllata", la clausola 'throws' e la conseguente gestione sono obbligatorie. Per le eccezioni "non controllate" decide il programmatore se mettere la clausola 'throws' e quindi gestirle, oppure no.

Se invece scrivo:

```
public static void main(String[] args){  
    m(); m3();  
    try {m2();}  
    catch (Throwable e) //catturo tutti gli errori o eccezioni  
        {System.out.println("Captured: " + e);}  
    System.out.println("Sono fuori dal try-catch!");  
}
```

```
// In questo caso il compilatore compila perché m() e m3() non dichiarano di lanciare l'eccezione con la 'throws' nella firma, quindi il compilatore accetta le due chiamate anche fuori dal 'try'. Tuttavia, durante l'esecuzione l'eccezione lanciata da m() o da m3() non viene gestita e si ha l'interruzione del programma. In particolare, la stampa System.out.println("Sono fuori dal try-catch!") non sarà eseguita. */  
} // fine class TestError
```

Possiamo definire noi stessi delle classi di eccezioni **MiaEccezione** estendendo una qualunque classe di eccezioni esistente: per le nostre eccezioni valgono le stesse regole che per le eccezioni di Java.

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, non sappiamo o non possiamo rimediare**. Quando sviluppiamo un programma utilizziamo gli assert per sollevare eccezioni non controllate, terminare il programma e tracciare l'errore: non catturiamo queste eccezioni. Le **eccezioni catturate** rappresentano le **procedure di emergenza** che uso quando incontro un'eccezione, voglio avvisare dell'errore ma **evitare di interrompere 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. Questa scelta è necessaria quando sto scrivendo un programma che *non può essere interrotto immediatamente in condizioni di sicurezza* (guida di un'automobile, di un aereo o di un drone), oppure che *fermato completamente produrrebbe danni economici* (un sistema di prenotazioni o di pagamenti elettronici), o quando *l'errore dipende dai dati inseriti* e deve essere prevista (chiediamo a una applicazione di aprire un file che non c'è). Come esempio, una eccezione non gestita condusse all'autodistruzione del razzo Ariane 501 (4 Giugno 1996). In questo corso non abbiamo il tempo di presentare questo ed altri esempi rilevanti di eccezioni, ma abbiamo tempo per alcuni semplici esempi.

Esempi di eccezioni non controllate (cattura possibile). Non è effettivamente **possibile né consigliabile** cercare di catturare tutte le eccezioni non controllate, **tipicamente esse provengono da errori logici nel programma che dovremmo piuttosto correggere**. Ma in casi particolari, se vogliamo possiamo farlo (o tentarlo). Per esempio, vediamo come sostituire l'uso di assert con il lancio di eccezioni non controllate e catturate. 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: proviamo entrambe le possibilità.

```

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 stampo
    {System.out.println("Catturata:" + 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 istruzione 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

    // Anche IllegalStateException e' una eccezione non controllata,
    // dunque se la sollevo NON sono obbligato ad aggiungere
    // "throws IllegalStateException" 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 "throws". Se lo facciamo, siamo obbligati a catturarla con try... catch nel momento in cui chiameremo il metodo.

```
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(). Se lo faccio sono obbligato a catturare
    // l'eccezione ogni volta che uso max()
    public int max() throws MaxSuAlberoVuoto {
        throw new MaxSuAlberoVuoto("max su Leaf");
    }
    // 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);}

    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 String toStringAux
(String prefix, String root, String left, String right)
{return prefix + root + "leaf"; }
}
// end class Leaf

```

Se volete potete provare ad usare questa versione della classe **Leaf**. In questo caso dovete aggiungere le classi **Tree**, **Branch** e **TestTree** viste nella lezione 17 sugli alberi di ricerca.

Esempi di eccezioni controllate (cattura necessaria). Proviamo a definire un'eccezione controllata, per impedire la creazione di una frazione di denominatore ≤ 0 , ma senza far cadere il programma quando questo capita. Per le eccezioni controllate siamo obbligati a scrivere "throws" nelle segnature dei metodi che le lanciano, e poi a catturarle nel codice che usa quei metodi. Definisco una classe di eccezioni controllate come sottoclassificazione della classe IOException di eccezioni controllate. Mettiamo come parametro dell'eccezione il denominatore della frazione rifiutata.

```

import java.util.*;
import java.io.*; //per IOException

class DenZeroException extends IOException{ //controllata
    private int den;
    public DenZeroException(int den){this.den=den;}
    public int getDen(){return den;}
}

class Frazione {
    private int num; private int den;
    // Invariante di classe: il denominatore deve essere > 0
    public Frazione(){num=1; den=1;} //costruttore pubblico:
    //restituisce un valore di default = 1/1

    // uso un costruttore 'private', puo' creare frazioni
    // non ben formate, ma non e' accessibile dall'esterno:
    private Frazione(int n, int d) {num = n; den = d;}
}

```

```

// Inserisco l'eccezione in un metodo 'create' pubblico
// (NON è un 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);
}
// 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".

// 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;}

} //end class Frazione

public class ProvaFrazione {
    public static void main(String[] args){
        Scanner scanner = new Scanner(System.in); //per leggere input
        boolean done = false;
        int n, d; Frazione f = new Frazione(); //f=default=1/1
        //f = Frazione.create(2,3);
        //NON OK: 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 (>0):");
                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");}
    /* In realta', IllegalStateException dovrebbe essere usata per indicare
     che lo stato del programma non consente l'uso di un dato metodo. */

    public static void main(String[] args) {
        try{
            m(); //otteniamo una IllegalStateException: prossima riga saltata
            System.out.println( "Impossibile! ho la risposta di m()");
        }

        catch (RuntimeException e) {
            //Questa clausola scatta perche'
            //RuntimeException include IllegalStateException
            System.out.println( "catturata IllegalStateException:\n" + e);
        }
        catch (Exception e) {
            //Questo catch verrebbe raggiunto da eccezioni diverse da
IllegalStateException
            System.out.println
                ("catturata eccezione (non IllegalStateException):\n" + e);
        }
        finally {
            //Qui volendo posso aggiungere una clausola
            //che sara' eseguita sia che arriviamo da try che da un catch
            System.out.println( "\n...posto per la clausola finale...\n");
        }
    }
} //end class TestTryCatch

```

Lezione 21

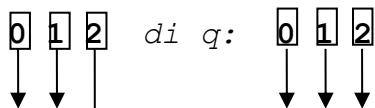
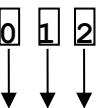
Esempi di esercizi di esame con soluzioni

Lezione 21. Forniamo esempi di esercizi di esami degli anni passati per gli argomenti del corso visti finora. Resta un solo argomento da svolgere, **le eccezioni**. Rimandiamo quest'ultimo argomento alla lezione successiva.

Lezione 21. Esercizio 1. Supponete sia già data la classe `Node<T>` di nodi generica. `Node<T>` rappresenta le liste concatenate su una classe `T` generica, e `null` denota la lista vuota. In `Node<T>` avete i metodi `T getElem()` e `Node<T> getNext()`. Come test di egualanza in `T`, vi chiediamo di usare `boolean x.equals(T y)`, che funziona per ogni oggetto `x≠null` di `T`. Supponete di avere due liste rappresentate da nodi `p, q` di `Node<T>`, e che gli elementi di `p,q` siano tutti `≠null` (dunque potete usare `equals`). Realizzate un metodo

```
public static <T> boolean inComune(Node<T> p, Node<T> q)
```

che restituisce `true` se e solo 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 i seguenti valori (nel disegno qui sotto numeriamo le posizioni da 0 e abbreviamo "posizione" con "pos.").

- posizioni di `p`:  di `q`: 
- **true** se `p` è **[5, 7]** e `q` è **[8, 7, 1]** (**7** in pos. 1 è in comune)
 - **true** se `p` è **[4, 3, 2]** e `q` è **[1, 2, 2]** (**2** in pos. 2 è in comune)
 - **true** se `p` è **[1, 2, 3]** e `q` è **[3, 2, 1]** (**2** in pos. 1 è in comune)
 - **false** se `p` è **[1, 2]** e `q` è **[2, 1]**
 - **false** se una lista è vuota oppure lo sono entrambe.

È obbligatorio definire `inComune` in modo ricorsivo.

Lezione 21. Esercizio 2. Siano 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("C.m2");obj.m1();}
}

```

Rispondete 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.m1();

```

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 21. Esercizio 3. Supponete sia già data la classe Node (concreta e senza uso di generici) per rappresentare nodi di liste concatenate su interi. In Node **null** denota la lista vuota, e abbiamo 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. Scrivete una **asserzione** che *describe sotto quali condizioni si può eseguire il metodo*. Aggiungete l'asserzione nella prima riga del metodo, come precondizione. L'asserzione deve consentire l'esecuzione del metodo se e solo l'esecuzione non solleva eccezioni. 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 21. Esercizio 4. Date le classi astratte ricorsive:

```
abstract class Tree {public abstract Tree insert(int elem);}

class Leaf extends Tree {
    public Tree insert(int elem)
        {return new Branch(elem, this, this);}
}

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 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");
    }
}
```

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 21. Soluzioni degli esercizi assegnati

Lezione 21. Soluzione Esercizio 1. Definiamo anche la classe `Node<T>`: in un esame dovete supporre che sia già data e non ridefinirla, se necessario ve la forniremo noi.

```
//Classe di nodi su T generica: un nodo e' null oppure e'
//un elemento di tipo T e l'indirizzo del prossimo nodo

public 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;}
}

public class Esercizio1 /* inComune(p,q) controlla se due liste p,q
su T hanno almeno un elemento in comune nella stessa posizione,
supponendo che gli elementi di p, q siano tutti !=null. Usa
x.equals(T y) per controllare se x,y sono uguali. */
{
    public static <T> boolean inComune(Node<T> p, Node<T> q){
        if ((p == null) || (q == null)) // non ci sono elementi in comune
            return false;
        else // p!=null e q!=null
        {
            T x=p.getElem(), y=q.getElem();
            Node<T> l=p.getNext(), m=q.getNext();
            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, se
necessario ve lo forniremo noi */

public static void main(String[] args)
{
    Node<String> p = null, q = null, r = null;
    String i="i", p1="a", p2="b"+i, q1="b"+i,q2="b"+i, r1="b"+i, r2="c";
    p = new Node<String>(p1,new Node<String>(p2,null)); //p={"a", "bi"}
```

```
q = new Node<String>(q1,new Node<String>(q2,null)); //q={"bi","bi"}  
r = new Node<String>(r1,new Node<String>(r2,null)); //r={"bi","c" }  
  
System.out.println("p={a,bi}, q={bi,bi}: bi in comune in pos. 1");  
System.out.println(" inComune(p,q) = " + inComune(p,q)+ "\n");  
System.out.println("q={bi,bi}, r={bi,c} : bi in comune in pos. 0");  
System.out.println(" inComune(q,r) = " + inComune(q,r)+ "\n");  
System.out.println("r={bi,c}, p={a,bi} : nessun elemento comune");  
System.out.println(" inComune(r,p) = " + inComune(r,p)+ "\n");  
}  
}
```

Se provate a sostituire **x.equals(y)** con **x==y** nel metodo inComune, è probabile che **il metodo non passi più i test**. Infatti, due stringhe "bi" in due diversi nodi potrebbero trovarsi in indirizzi diversi, ed essere considerate diverse dal test **x==y**.

Lezione 21. 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 Esercizio2{
    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.m1(); */

/* 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); */  
}  
}  
// end class Esercizio 2
```

Lezione 21. 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 su interi, 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)
                //se p ha un elem negativo che non e' seguito da elementi allora:
                return false;
            p=p.getNext();
        }
        /*Se il while finisce, allora ogni elemento
        negativo in p e' seguito da un elemento, allora: */
        return true;
    }

    /* DESCRIZIONE metodo(p): per ogni elemento negativo in p, il metodo
       cerca l'elemento seguente e lo elimina se positivo, altrimenti non lo
       elimina, in ogni caso continua. Quindi metodo(p) esamina tutti gli
       elementi negativi nella versione originaria di p, e 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();
    }
}

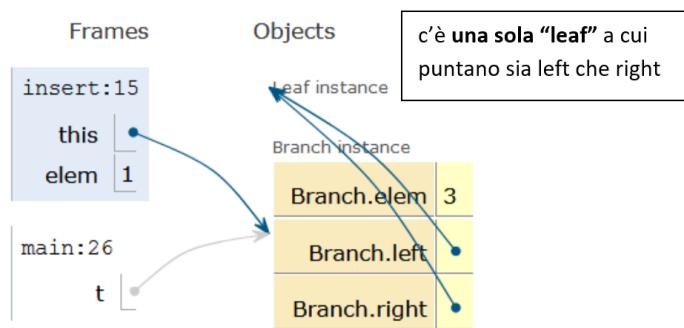
public static void main(String[] args)
{
    Node p = new Node(0, new Node(-1,null));
//p={0,-1}
    Node q = new Node(0, new Node(-1,new Node(+1,null)));
//q={0,-1,+1}
    System.out.println(" ok(p)=" + ok(p) + " ok(q)=" + ok(q));
    System.out.println(" p =");
    scriviOutput(p);

    if (!ok(p)) //p non viene accettata da ok(p) perche':
        System.out.println("l'elemento -1 non ha successore in p");
//metodo(p); //NO: se lancio metodo(p) scatta l'asserzione

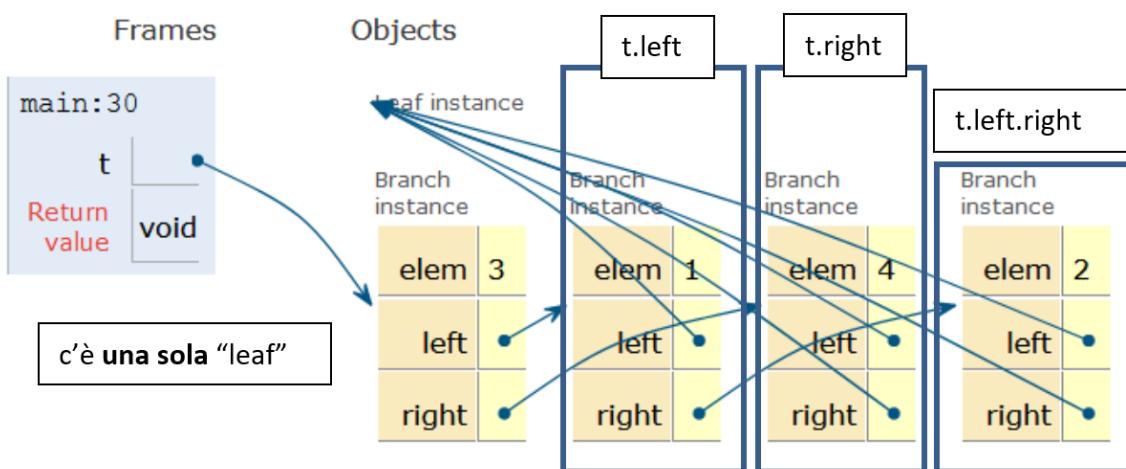
    System.out.println(" q =");
    scriviOutput(q);
    metodo(q);
//metodo(q) cancella +1 perche' +1 segue -1 in q
    System.out.println( "dopo metodo(q) : q=");
    scriviOutput(q);
}
}
```

Lezione 21. Soluzione Esercizio 4.

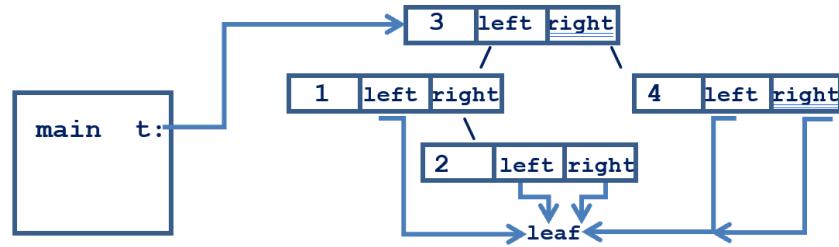
Il **check point 2** scatta la prima volta che dobbiamo aggiungere un nodo a un albero che ne contiene già uno, ma non l'abbiamo ancora fatto. Dunque abbiamo eseguito `t = t.insert(3);` e creato un albero con il solo nodo 3. Il figlio sinistro e destro sono rappresentati con la stessa foglia per risparmiare spazio. 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`. L'inserimento ordinato ha messo 3 come radice di `t`, 1 e 4 come figli sinistro e destro. Il 2, inserito per ultimo, è stato posto tra 1 e 3, come figlio di 1. Infine, l'albero ha cinque "foglie", tutte rappresentate con lo stesso elemento di Leaf per ragioni di spazio: infatti nel disegno **vediamo cinque frecce verso la stessa "foglia"**. Ecco la situazione di Stack e Heap come viene descritta da un Java visualizer.



Qui sotto lo stesso disegno reso più leggibile, con t.left a sinistra e t.right a destra, e con l'unica foglia disegnata in basso.



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.

Consideriamo (in versione giocattolo) un passo preliminare della compilazione, il problema di raggruppare in “**token**” i caratteri che compongono una espressione di un linguaggio di programmazione, **sollevarlo una input/output exception** in caso di fallimento. Un token è la coppia **<tag, text>** di una “tag”, una etichetta che dice a cosa serve una parte di una espressione, e del testo che compone quella parte di espressione. Consideriamo le seguenti tag:

```
//Tag.java
public enum Tag {
    //etichette che indicano l'uso di parti di una espressione
    NUM,                                // numero
    ID,                                   // identificatore (costante o variabile)
    PLUS,MINUS,TIMES,DIVIDE,           // diverse operazioni ...
    EOF}                                  // End Of File: un token per terminare
```

Isolare un testo in “token” è il primo passo per capirlo. Nel file Tag.java abbiamo inserito una “tag” per i numeri, i nomi delle operazioni e così via. Per esempio: nell’espressione

$$(bMin+bMag) *h/2;$$

possiamo isolare i seguenti “token”:

$$<\text{LPAR}, "()">, \quad <\text{ID}, "bMin">, \quad <\text{PLUS}, "+">, \quad <\text{ID}, "bMag">, \quad ...$$

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

    //Producgo 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);
    }
}

```

Definiamo ora una classe “Lexer” di analizzatori di testo *lexer*. Un lexer legge da tastiera un testo, che rappresenta una espressione fatta con numeri e le quattro operazioni, restituisce il primo token che trova in questo testo, e si ferma al primo carattere dopo il token. Un lexer restituisce il token speciale “EOF” quando raggiunge la fine del testo, e restituisce una eccezione quando trova un carattere non previsto. In questo caso il lexer lancia una eccezione controllata ***IOException***, che siamo obbligati poi a catturare e a trattare in qualche modo.

Attributi e metodi di Lexer sono:

1. **char peek.** È l'unico attributo (privato) di un lexer, 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 finché 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, che siamo obbligati a catturare. `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()`.

```
//Lexer.java. Classe degli analizzatori di testo per espressioni
import java.io.*; // Per avere la classe IOException, che altrimenti
//devo descrivere come: java.io.IOException

public class Lexer {
    private 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 EOF=-1.
    }

    public Token scan() throws IOException //eccezione da catturare
    {
        //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' +*-/");
}
} //end switch
}

```

//NOTA. Un metodo che usa un metodo che solleva eccezioni controllate
//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,
//inserisce un messaggio e esce dal while

// main di prova
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)

```

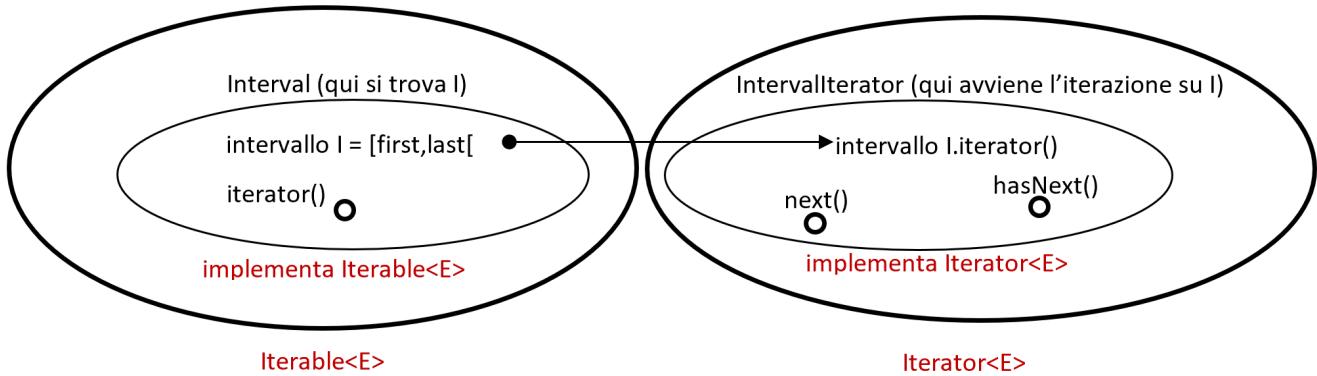
```

/** Uso scan() finche' arrivo a fine testo. Non stampo il token di
fine testo. */
try
{t = lex.scan();
 System.out.println( "Token: " + t);
}
catch(IOException e)
/** quando viene inserito un carattere non previsto scrivo un
messaggio di errore e poi termine il ciclo con un tag EOF */
{
    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[` che implementa `Iterable<Integer>`. La `[first, last[` è la **lista di primo nodo first e senza i nodi della lista dal nodo last in poi** (`last` è escluso), e con tutti i nodi nella lista da `first` se `last` non vi compare. **Esempi.** Se `first={1,2,3}` allora `[first,null[= {1,2,3}`, `[first,{4}[= {1,2,3}` e `[first,first[= {}`. Se `last = terzo nodo di first`, allora `[first,last[= {1,2}` (scarto il terzo nodo, contenente il numero 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.

Per ottenere questo effetto, definiamo una seconda classe `IntervalIterator`, e implementiamo in queste due classi **le interfacce `Iterable<Integer>` e `Iterator<Integer>`**, seguendo lo schema già visto nella Lezione 19, sempre con **`E = Integer`**. Il prossimo disegno riassume la situazione.



```

import java.util.*; //per le interfacce Iterable<T> e Iterator<T>

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 Iterator<Integer> 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()
    {
        if(first==last)
            throw new NoSuchElementException();
        Node temp = first;
        first = first.next;
        return temp.elem;
    }
}

```

```

}

public Integer next() //portiamo avanti first senza cambiare last
//e restituiamo il contenuto del first originario
{
    assert first!=last && first!=null;
    int elem=first.getElem();
    first=first.getNext();
    return elem;
}
}

// TestIterator.java. Scrivo una classe per
// sperimentare 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" );
        //nodi di p fino alla fine
        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[ = {}" ); // nessun nodo di 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 (30 minuti). Siano date le classi (incomplete) :

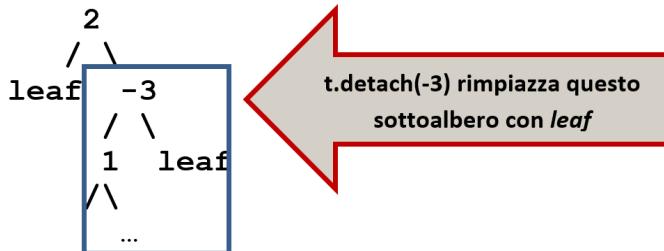
```
abstract class Tree<T> {public abstract Tree<T> detach(T x);}

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 (obbligatoriamente **ricorsive**) 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**. `detach(x)` deve funzionare anche quando `x=null`. Se `x` non è presente, il metodo deve restituire l'albero invariato. **Esempio.** Sia `t` un albero di radice 2, figlio sinistro una foglia e figlio destro di radice -3.



`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=elem` in `T` dovete usare "**equals**" controllando prima che `x!=null`, e "`==`" se `x=null`.

Lezione 23. Esercizio 2 (30 minuti).

```

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 (30 minuti). Sia dato il seguente metodo:

```
class Node<T> {public T elem; public Node<T> next;
    public Node(T elem, Node<T> next) {this.elem=elem;this.next=next;}
    public T      getElem()           {return elem;}
    public Node<T> getNext()         {return next;}
    public void    setElem(T elem)    {this.elem=elem;}
    public void    setNext(Node<T> next) {this.next=next;}}
```

```
public class Esercizio3{
    public static <T extends Comparable<T>> void metodo(Node<T> p, T x)
    {
        while (x.compareTo(p.getElem()) < 0) // x < p.getElem() (in T)
            p = p.getNext();
        p.setNext(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 (60 minuti). (*Questo esercizio richiede un po' più di tempo di un normale esercizio di compito). Si disegnino Stack e Heap la prima volta che viene raggiunto il **checkpoint 1** e il **checkpoint 2** durante l'esecuzione del 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 Esercizio4 {
    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 (diverso da null) di Nil, allora eseguendo **this.reverse(prev)** otteniamo che ora la freccia che esce da 1 punta a prev, quale che sia la dimensione della lista prev, e la freccia che esce da 2 punta a 1. Ecco il disegno:



Lezione 23. Soluzione esercizio 1.

L'idea per una soluzione ricorsiva è restituire "leaf" ogni volta che incontriamo un albero non vuoto di radice uguale ad `x`. Se l'albero è una foglia allora `detach` restituisce la stessa foglia (per non moltiplicare inutilmente le foglie), se non è vuoto ma ha radice $\neq x$ allora modifichiamo i sottoalberi sinistro e destro e restituiamo l'albero di partenza così modificato.

Dobbiamo ricordarci di sostituire i sottoalberi sinistro e destro originali con i nuovi con `left = left.detach(x)` e `right = right.detach(x)`, altrimenti la modifica non viene memorizzata.

Per controllare se `x = elem`, dobbiamo usare `==` quando `x` è null, `x.equals(elem)` altrimenti, quindi scrivere un test

```
(x==null) && (x==elem) || (x!=null) &&x.equals(elem)
```

Attenti all'ordine con cui scrivete il test per `x = elem`: bisogna che `x.equals(elem)` sia eseguito solo quando siamo sicuri che `x \neq null`, altrimenti inviare un metodo dinamico a null solleva una `NullPointerException`. Quindi `x.equals(elem)` deve essere eseguito dopo aver eseguito un test (`x!=null`) (come avviene qui sopra).

Abbiamo aggiunto un metodo `toString()` per stampare gli alberi e un main di prova, con lo stesso albero t citato nel testo dell'esercizio.

```
abstract class Tree<T> {public abstract Tree<T> detach(T x);}

class Leaf<T> extends Tree<T> {
    public String toString(){return "leaf";}

    public Tree<T> detach(T x) {return this;}
} /*nella classe Leaf abbiamo: this = albero vuoto sempre */

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 String toString()
        {return "(" + left + " " + elem + " " + 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;}
}
}

//main di prova con l'albero t citato nel testo dell'esercizio
public class Esercizio1
{
    public static void main(String[] args)
    {
        Tree<Integer> leaf = new Leaf<>(),
        t = new Branch<>(2,
                           leaf,
                           new Branch<>(-3,
                                         new Branch<>(1,leaf,leaf),
                                         leaf)),
        u = new Branch<>(null,
                           leaf,
                           new Branch<>(-3,
                                         new Branch<>(1,leaf,leaf),
                                         leaf)),
        v = new Branch<>(null,
                           leaf,
                           new Branch<>(-3,
                                         new Branch<>(1,leaf,leaf),
                                         leaf));

        System.out.println( "\n t = " + t);
        System.out.println( "    t.detach(-3) = " + t.detach(-3));
        System.out.println( "\n u = " + u);
        System.out.println( "    u.detach( 1) = " + u.detach( 1));
        System.out.println( "\n v = " + v);
        System.out.println( "    v.detach(10) = " + v.detach(10));
    }
}

```

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 è tenuta a fornire una implementazione per tutti i metodi elencati nelle interfacce che implementa. Inoltre il metodo m1 in C è astratto, dunque le sottoclassi di C (in particolare, D) non lo possono invocare con un "super" e dunque non ne hanno bisogno.

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 inclusa in I: downcast compila
        ((D) obj2).m2(); //D inclusa in I: downcast corretto
/*Risultato: D.m2 D.m1 */

        System.out.println( " Domanda 3");
        J obj3 = new D();//D inclusa in J: downcast compila
        C x = (C) obj3; //J, C hanno D in comune: downcast compila
        x.m1(new D()); //Durante esecuzione: x si trova in D
/*Risultato:
        D.m2 // chiamata di m1 a m2: il resto di m1 e' in sospeso
        D.m1 // chiamata di m2 a m1: m1 si trova sullo stack due volte
        D.m1 // fine prima delle chiamate di m1 in sospeso
*/
/* System.out.println( " Domanda 4");
    C obj4 = new D();
    obj4.m1(obj4); */
}

```

/* Risposta. obj4, per essere inserito come argomento di m1,
dovrebbe avere un tipo apparente che si converte a J, ma obj4 ha un
tipo apparente C, che si converte ad I, ma non si converte a J.
Compilando, 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)
```

Se invece si scrive obj4.m1((J)obj4); l'istruzione e' corretta: C, J
hanno una classe D in comune, e durante l'esecuzione obj4 sta in D
*/

```
}
```

Lezione 23. Soluzione Esercizio 3.

Risposta 1. Il metodo confronta x con il primo elemento di p , poi prosegue leggendo il prossimo elemento della lista p finché trova in p elementi $e=p.getElem()$ tali che $x < e$. Dunque il metodo lancia un'eccezione se tolti dalla lista p tutti gli elementi e tali che ($x < e$) si ottiene la lista vuota (in particolare se $p=null$), oppure se $x=null$ (non possiamo mandare il metodo equals a null). 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 $e' \leq x$.

Ora scriviamo l'asserzione richiesta.

```
class Node<T> {
    public T elem;
    public Node<T> next;
    public Node(T elem, Node<T> next){this.elem=elem;this.next=next;}
    public T      getElem()           {return elem;}
    public Node<T> getNext()         {return next;}
    public void   setElem(T elem)    {this.elem=elem;}
    public void   setNext(Node<T> next) {this.next=next;}
}

public class Esercizio3{
    public static void scriviOutput(Node p){
        while(p!=null)
            {System.out.println(""+p.getElem()); p=p.getNext();}
    }

    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.getElem()) < 0) //x < p.getElem() (in T)
            p=p.getNext();
        p.setNext(null);
    }
}
```

```

public static <T extends Comparable<T>> boolean OK(Node<T> p, T x) {
    if (x==null) return false;
    // adesso sappiamo che x!=null
    while ((p!=null) && (x.compareTo(p.getElem()) < 0))
    // x<p.getElem() e p!=null, dunque p.getElem() esiste
        p=p.getNext();
    return p!=null; /* se sono arrivato a p==null allora metodo produce
una NullPointerException. Altrimenti metodo tronca al primo elemento
e di p tale che sia x>=e */
}

//main di prova per il metodo OK
public static void main(String[] args) {
    Node<Integer> p = new Node<>(30,
                                    new Node<>(20, new Node<>(10, null)));
    System.out.println("Sia p={30,20,10}");scriviOutput(p);
    System.out.println("Esempi di valori di OK:");
    System.out.println(" OK(null,30)=" + OK(null,30));
    //OK(null,30)=false
    System.out.println(" OK(p,null)=" + OK(p,null));
    //OK(p,null)=false

    System.out.print("\n Uso metodo per ");
    System.out.println("eliminare gli elementi di p dopo il primo
<=10");
    System.out.println("Data che p={30,20,10}, p resta uguale");
    System.out.println(" OK(p,10)=" + OK(p,10)); //OK(p,10)=true
    System.out.println(" Applico metodo(p,10): ottengo");
    metodo(p,10); scriviOutput(p);

    System.out.print("\n Uso metodo per ");
    System.out.println("eliminare gli elementi di p dopo il primo
<=20");
    System.out.println("Data che p={30,20,10}, p diventa {30,20}");
    System.out.println(" OK(p,20)=" + OK(p,20)); //OK(p,20)=true
    System.out.println(" Applico metodo(p,20): ottengo");
    metodo(p,20); scriviOutput(p);

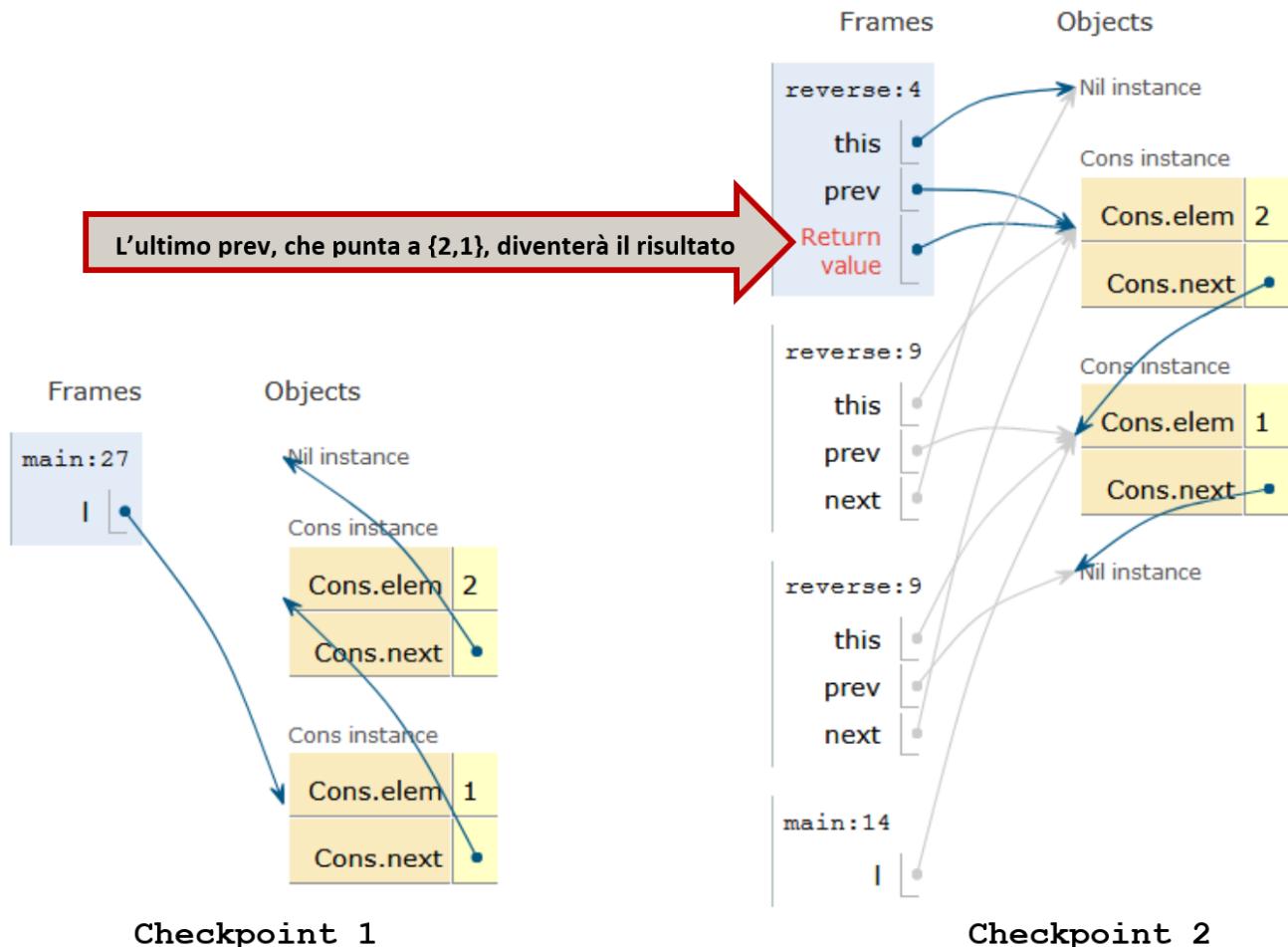
    System.out.print("\n Uso metodo per ");
    System.out.println("eliminare gli elementi di p dopo il primo
<=10");
}

```

```
System.out.println("Dato che p={30,20}, ottengo una eccezione");
System.out.println(" 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. (*Questo esercizio richiede un po' più di tempo di un normale esercizio di compito). 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. Arriviamo al **checkpoint 2** quando usiamo il metodo per la classe `Nil`, dunque quando `this=nil` (quando la lista arriva alla fine). 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.



Alla fine `prev` vale `{2,1}`.

Lezione 24

Visita di un albero in pre-, in-, post-ordine e per livelli

In questa lezione presentiamo degli esercizi sugli alberi. Riguardano dei metodi che visitano un albero secondo diversi ordini: pre-ordine, in-ordine, post-ordine e per livelli. Durante una visita possiamo eseguire la stessa operazione su tutti i nodi dell'albero, per esempio una stampa. Inoltre calcoleremo anche il numero di foglie in ogni "livello" dell'albero. Un "livello" è fatto di tutti i nodi alla stessa distanza dalla radice. Le visite in pre-, in-, post-ordine sono definite ricorsivamente, come segue.

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 esplorano completamente un ramo prima di passare al ramo accanto. 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 via.

Questi ordini sono utilizzati in diversi algoritmi che coinvolgono gli alberi. Si tratta di argomenti che non fanno parte di ProgII, ma facciamo comunque un breve 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** (occorre qualche passo in più, per ritornare ogni volta da un nodo-figlio alla radice). Per **disegnare un albero** da sinistra a destra facciamo comparire i nodi in **in-ordine** (*si noti che una visita in-ordine di un albero di ricerca visita l'albero seguendo la relazione d'ordine codificata nell'albero*). 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();

    /* Le tre visite che seguono, in pre-order, in-order e post-
       order, sono dette visite "in profondità".
    */
    //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();

    //restituisce il livello numero n sotto forma di stringa
    public abstract String livello(int n);

    /* La visita che segue, per livelli, è anche detta visita "in
       ampiezza". */
    //stampa un albero per livelli
    public abstract void livello();

    //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 String toStringAux
        (String prefix, String root, String left, String right);
    //gestisce la parte NON pubblica della stampa.
    //Non forniamo spiegazioni sul suo funzionamento, non e'
    //essenziale

    public String toString()
        {return toStringAux("", "__", "    ", "    ");}

    // metodo pubblico di stampa, dall'alto verso il basso, con i
    // sottoalberi disegnati piu' a destra dell'albero di cui fan

```

```

// parte. Disegno bidimensionale fatto con soli caratteri ascii.
}

//Leaf.java gli oggetti (non null) di questa classe rappresentano un
//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 "";}
    //una leaf non contiene un elemento
    //per questo si restituisce la stringa vuota

    public void livello(){}

    public int leavesAt(int n){if (n==0) return 1; else return 0; }

    //Metodo che gestisce la parte NON pubblica della stampa.
    //Non forniamo spiegazioni sul suo funzionamento.
    protected String toStringAux
        (String prefix, String root, String left, String right)
        {return prefix + root + "leaf";}
}

//Branch.java
public class Branch extends Tree {
    private int elem;    //radice
    private Tree left, right;

    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 in 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.
//Non forniamo spiegazioni sul suo funzionamento, non e' essenziale.
protected String toStringAux
(String prefix, String root, String left, String right){
    return this.left.toStringAux(prefix+left, " /", " ", " |")
        + "\n" + prefix + root + "[" + elem + "]"
        + "\n" + this.right.toStringAux(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). Radice 1,
figlio sinistro 2, figlio destro 3, che ha figlio destro 72, che ha
figlio destro 9.

      /leaf
    /[2]
    | \leaf
_____[1]
    | /leaf
    \[3]
      | /leaf
      \[72]
        | /leaf
        \[9]
          \leaf

```

Rappresentazione per livelli dello stesso albero t. Al livello 0 vediamo 1, al livello 1 vediamo i figli 2 e 3, al livello 2 vediamo tre foglie e il nipote 72, al livello 3 vediamo il pronipote 9 e una foglia, al livello 4 due foglie. I successivi livelli sono vuoti.

Livello

Albero t (radice in alto, figli da sinistra)

```

0           1
|           / \ 
1           2   3
| \         / \ 
2   leaf   leaf  72
| \         / \ 
3   leaf   9
| \         / \ 
4   leaf   leaf
  
```

```

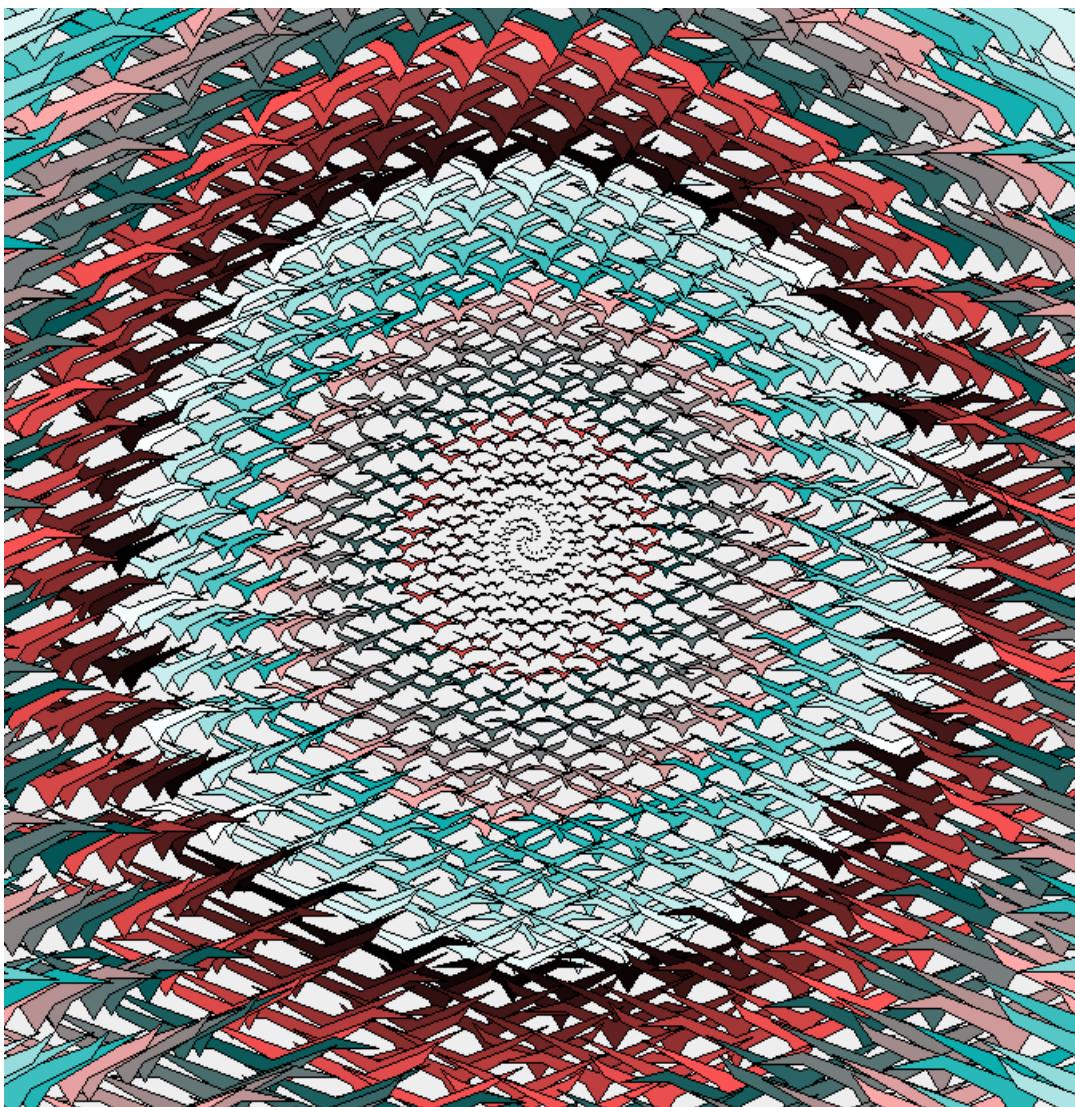
t in pre-order:  1 2 3 72 9
t in in-order:   2 1 3 72 9
t in post-order: 2 9 72 3 1
t per livelli:   1 2 3 72 9
  
```

```

*/
System.out.println( "\n L'albero t: \n" + t);
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( "\n 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 */
}
}
  
```

**Fine del Corso:
Programmazione II
Corso di Laurea in Informatica
Università di Torino**



Una spirale disegnata con la classe Java "Figure" vista a lezione