

# Insegnamento

## “Principi di Programmazione

### Orientata agli Oggetti”

### - Dispense delle lezioni -

Università degli Studi di Torino  
Corso di Laurea in Informatica  
a.a. 2024/25

Docenti: Liliana Ardissono, Viviana Bono,  
Sara Capecchi, Luigi Di Caro, Jeremy  
Sproston

Parte del materiale è tratto dalle  
Dispense di Programmazione II - Java

- 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.

# Indice delle Lezioni

|  |     |
|--|-----|
| <u>Indice delle Lezioni</u>  | 2   |
| <u>Presentazione dell'insegnamento</u>   | 4   |
| <u>"Principi di Programmazione Orientata agli Oggetti" (PPOO) - a.a. 2024/25</u>   | 4   |
| <u>Testi consigliati</u>   | 4   |
| <u>Scheda ufficiale del corso (con programma e modalità d'esame)</u>   | 4   |
| <u>Principi di Programmazione Orientata agli Oggetti - Corso di laurea in Informatica - Università degli Studi di Torino</u> | 4   |
| <u>Organizzazione e distribuzione del materiale didattico per Principi di Programmazione Orientata agli Oggetti II A, B</u>  | 4   |
| <u>Ambiente di sviluppo e visualizzatore Java</u>  | 5   |
| <u>Lezione 01</u>  | 7   |
| <u>Introduzione alla programmazione orientata agli oggetti</u>   | 7   |
| <u>Lezione 02</u>  | 28  |
| <u>Classi, attributi e metodi pubblici e privati</u>   | 28  |
| <u>Lezione 03</u>  | 46  |
| <u>Attributi e metodi privati, get e set</u>   | 46  |
| <u>Lezione 04</u>  | 59  |
| <u>Assegnazioni di oggetti, metodo equals, costruttori</u>   | 59  |
| <u>Lezione 05</u>  | 73  |
| <u>La classe "Stack". Chiamate di metodi</u>   | 73  |
| <u>Lezione 06</u>  | 82  |
| <u>Modelli di oggetti reali. Information Hiding</u>  | 82  |
| <u>Lezione 07</u>  | 92  |
| <u>Classi di array di oggetti</u>  | 92  |
| <u>Lezione 08</u>  | 105 |
| <u>Diagrammi UML e array estendibili</u>   | 105 |
| <u>Lezione 09</u>  | 114 |
| <u>Security Leak.</u>  | 114 |
| <u>Le classi Node e DynamicStack</u>   | 114 |
| <u>Lezione 10</u>  | 127 |
| <u>Metodi statici per la classe Node</u>   | 127 |
| <u>Lezione 11</u>  | 138 |
| <u>Classi generiche: coppie, nodi e pile</u>   | 138 |
| <u>Lezione 12</u>  | 151 |
| <u>Ereditarietà e assert</u>   | 151 |
| <u>Lezione 13</u>  | 168 |
| <u>Estensioni ripetute di classi</u>   | 168 |
| <u>Lezione 14</u>  | 177 |

|  |     |
|--|-----|
| <u>Tipo esatto e binding dinamico</u>  | 177 |
| <u>Lezione 15</u>  | 191 |
| <u>Esempi di ereditarietà. Array come liste</u>  | 191 |
| <u>Lezione 16</u>  | 206 |
| <u>Classi astratte di figure</u>   | 206 |
| <u>Lezione 17</u>  | 219 |
| <u>La classe astratta ricorsiva degli alberi di ricerca</u>  | 219 |
| <u>Lezione 18</u>  | 230 |
| <u>Visita di un albero binario in pre-, in-, post-ordine e per livelli. Note sul metodo equals()</u> | 230 |
| <u>Lezione 19</u>  | 245 |
| <u>Interfacce, generici vincolati e alberi di ricerca</u>  | 245 |
| <u>Lezione 20</u>  | 261 |
| <u>Interfacce Comparable, Iterator e Iterable</u>  | 261 |
| <u>Lezione 21</u>  | 275 |
| <u>Eccezioni controllate e non controllate</u>   | 275 |
| <u>Lezione 22</u>  | 292 |
| <u>Cenni alla progettazione orientata agli oggetti: i design pattern Singleton, Composite, State</u> | 292 |
| <u>Lezione 23</u>  | 315 |
| <u>Esercizi</u>  | 315 |
| <u>Appendice</u>   | 316 |
| <u>(NON FA PARTE DEL PROGRAMMA DEL CORSO)</u>  | 316 |

# **Presentazione dell'insegnamento "Principi di Programmazione Orientata agli Oggetti" (PPOO) - a.a. 2024/25**

Nel 2024/2025 questo insegnamento si svolge in 48 ore (24 lezioni di 2 ore) ed è associato a 30 ore di laboratorio (10 lezioni di 3 ore, su due turni, T1 e T2).

## **Testi consigliati**

- Paul J. Deitel - Harvey M. Deitel. Programmare in Java - 11/Ed. MyLab, 2020.
- M. Fowler. UML Distilled - 4/Ed. Pearson, 2017.
- M. Naftalin, P. Wadler. Java Generics and Collections. O'Reilly, 2006.

## **Scheda ufficiale del corso (con programma e modalità d'esame)**

[Principi di Programmazione Orientata agli Oggetti - Corso di laurea in Informatica - Università degli Studi di Torino](#)

## **Organizzazione e distribuzione del materiale didattico per Principi di Programmazione Orientata agli Oggetti II A, B**

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

Il materiale didattico è distribuito come segue:

- **Queste dispense saranno work-in-progress per tutta la durata del corso. Verranno indicati i capitoli considerati stabili volta per volta su I-learn. Tuttavia, ci riserviamo di fare cambiamenti anche ai capitoli già affrontati, però segnalando chiaramente i punti di cambiamento.**
- Gli argomenti per il corso e il laboratorio sono di norma pubblicati sul sito I-learn corrispondente all'inizio di ogni settimana.
- La **versione del codice sorgente** visto a lezione, **se modificata durante la lezione stessa**, è di norma pubblicata entro il termine di ogni settimana (in

particolare, il codice è ***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.

Un'ultima osservazione: il codice delle lezioni è ***sovra-commentato*** (i commenti fanno parte del materiale della lezione), mentre lo stesso codice quando rivisto nel corso di laboratorio sarà a volte ***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.

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:

#### [Java Visualizer](#)

(**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.



L'icona di Java: una tazzina di caffè

# **Lezione 01**

## **Introduzione alla programmazione orientata agli oggetti**

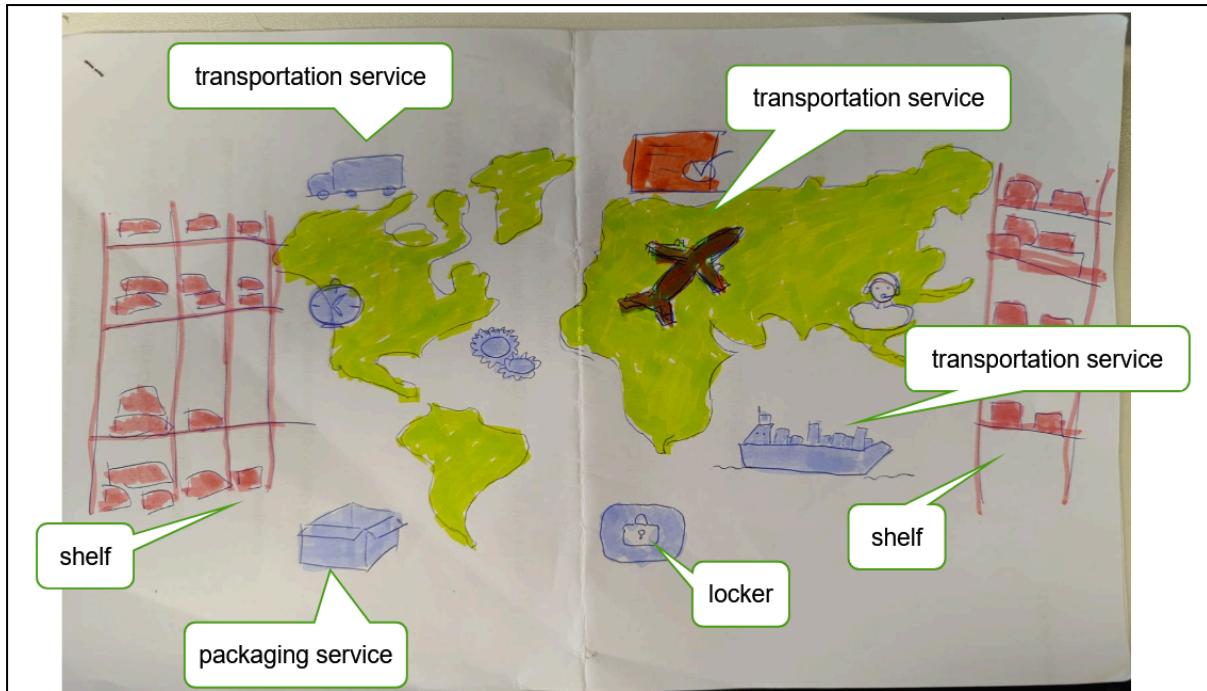
### **Introduzione alla Programmazione orientata agli oggetti.**

La progettazione orientata agli oggetti (OOD) e la programmazione (OOP) sono state introdotte per gestire la complessità del software. Con la diffusione dell'informatica in ogni settore e la conseguente necessità di sviluppare nuove applicazioni, e l'evoluzione dell'hardware, le dimensioni dei programmi (in termini di linee di codice) sono aumentate drasticamente, determinando criticità nello sviluppo, nel debug e nella manutenzione del software. Per questo motivo, sono stati ideati **nuovi paradigmi di programmazione per aiutare i programmati a sviluppare con maggiore facilità il software.**

Questo capitolo ha lo scopo di fornire un'intuizione sui concetti chiave dell'**Object Oriented Programming (OOP; Programmazione orientata agli oggetti)** che verranno studiati in dettaglio nel resto dell'insegnamento. Il suo scopo è aiutare i programmati che hanno familiarità con linguaggi imperativi, come il C, a passare alla programmazione in linguaggi orientati agli oggetti, come Java. La difficoltà del cambio di paradigma da imperativo a orientato agli oggetti è la comprensione di:

- come progettare un'architettura modulare di un'applicazione software, esplicitando i servizi e i sottoservizi su cui si basa, e
- come questi servizi debbano collaborare per implementare congiuntamente la funzionalità complessiva che l'applicazione fornisce (ad esempio, un servizio di prenotazione alberghiera). Intuitivamente, **gli oggetti, nella OOP, sono fornitori di servizi più o meno complessi che interagiscono tra loro per ottenere il servizio composito.**

Come metafora per rappresentare la collaborazione tra oggetti, si pensi all'insieme di servizi che collaborano alla gestione di una catena di fornitura. Il disegno seguente lo schematizza.



I servizi possono essere distribuiti in tutto il mondo e interagire tra loro per gestire la catena di fornitura e portare i prodotti sugli scaffali. Ogni servizio è gestito da un sistema diverso (scaffali, pianificazione delle attività, logistica, trasporti aerei e terrestri, ecc.). Alcuni servizi sono gestiti da utenti umani attraverso le loro interfacce utente.

La progettazione e la programmazione orientata agli oggetti è una tecnologia chiave per lo sviluppo e la gestione di un'architettura così complessa. In particolare, vorremmo sottolineare fin da ora come i linguaggi orientati agli oggetti permettano un passaggio molto più diretto dalla progettazione all'implementazione.

**La maggior parte del testo di questo capitolo è estratto dal libro "Object-Oriented analysis and design", 3rd Edition, by Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Ph.D., Jim Conallen, Kelli A. Houston, Addison Wesley,**

<https://dl.acm.org/doi/10.5555/1407387>. Il resto è stato recuperato da internet, o prodotto dagli autori delle dispense, o da altri docenti del Dipartimento di Informatica dell'Università di Torino (Giovanna Petrone, Annamaria Goy). La traduzione dal testo inglese alla lingua italiana è stata effettuata adattando il testo prodotto da DeepL.com (versione gratuita, <https://www.deepl.com/it/translator>).

**"Più il sistema è complesso, più è esposto a una rottura totale"**<sup>1</sup>.

Raramente un costruttore penserebbe di aggiungere un nuovo piano interrato a un edificio esistente di 100 piani. Sarebbe molto

---

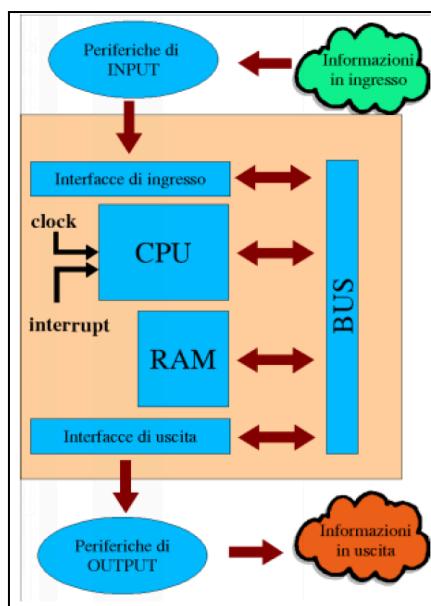
<sup>1</sup> Peter, L. 1986. The Peter Pyramid. New York, NY: William Morrow, p. 153.

*costoso e senza dubbio si andrebbe incontro a un guasto. Sorprendentemente, gli utenti dei sistemi software raramente ci pensano due volte a chiedere modifiche equivalenti. Inoltre, sostengono, si tratta solo di una semplice questione di programmazione. La nostra incapacità di padroneggiare la complessità del software si traduce in progetti in ritardo, fuori budget e carenti rispetto ai requisiti dichiarati.*

Poiché il problema di fondo deriva dalla complessità intrinseca del software, il nostro suggerimento è di studiare innanzitutto come sono organizzati i sistemi complessi in altre discipline.

### La complessità e l'organizzazione gerarchica dei sistemi complessi

Osserviamo che, in domini molto eterogenei, i sistemi complessi hanno una natura gerarchica. Come mostriamo nel seguito, questa proprietà è comune agli artefatti costruiti dall'uomo nonché ai sistemi naturali.



Prendiamo come esempio il personal computer (PC, dominio hardware). I PC sono dispositivi di moderata complessità. La maggior parte di essi è composta dagli stessi elementi principali: un'unità di elaborazione centrale (CPU), un monitor, una tastiera e un dispositivo di archiviazione secondario, solitamente un'unità CD o DVD e un disco rigido. Possiamo prendere una qualsiasi di queste parti e scomporla ulteriormente. Ad esempio, una CPU comprende la memoria primaria, una unità aritmetica/logica (ALU) e un bus a cui sono collegati i dispositivi periferici.

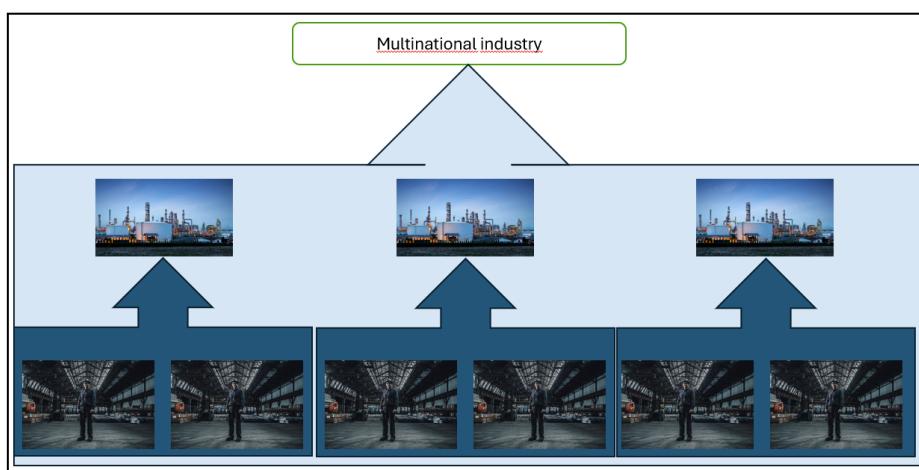
**Un personal computer funziona correttamente solo grazie all'attività collaborativa di ciascuna delle sue parti principali. Insieme, queste parti separate formano logicamente un insieme.**

Possiamo ragionare sul funzionamento di un computer solo perché possiamo scomporlo in parti che possiamo studiare separatamente. Così, possiamo studiare il funzionamento di un monitor indipendentemente dal funzionamento del disco rigido. Allo stesso modo, possiamo studiare l'ALU senza considerare il sottosistema della memoria primaria.

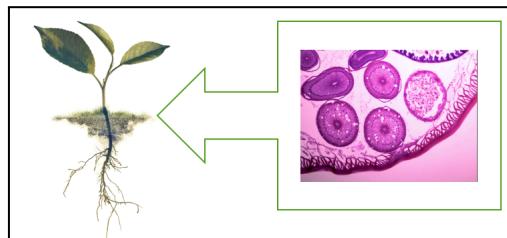
I sistemi complessi non solo sono gerarchici, ma i livelli di questa gerarchia rappresentano diversi livelli di astrazione, ognuno costruito sull'altro e ognuno comprensibile da solo. **A ogni livello di astrazione, troviamo un insieme di dispositivi che collaborano per fornire servizi ai livelli superiori.**

**Si sceglie un determinato livello di astrazione per soddisfare le proprie esigenze.** Per esempio, se stiamo cercando di individuare un problema di latenza temporale nella memoria primaria, potremmo guardare correttamente all'architettura a livello di porta (gate) del computer, ma questo livello di astrazione sarebbe inappropriato se stessimo cercando di trovare la fonte di un problema in un'applicazione di foglio elettronico.

Un altro esempio di sistemi complessi è la struttura delle istituzioni sociali. Gruppi di persone si uniscono per portare a termine compiti che non possono essere svolti da singoli individui. Man mano che le organizzazioni si ingrandiscono, emerge una gerarchia distinta. Le multinazionali (multinational industry nella figura) contengono aziende, che a loro volta sono costituite da divisioni, che a loro volta contengono filiali, che comprendono uffici locali, e così via. Si noti anche che questi componenti svolgono ruoli specifici all'interno delle organizzazioni (proprio come i singoli dipendenti svolgono un ruolo distinto nel loro ufficio).



**NOTA: Le relazioni tra le varie parti di una grande organizzazione sono proprio come quelle che si trovano tra i componenti di un computer o di altri sistemi complessi come una pianta (foglie, radici, ecc. e, a livelli più fini, cellule) o un essere umano (polmoni, cuore, ecc. e, a livelli più fini, cellule).**



### Il ruolo della scomposizione

**"La tecnica per dominare la complessità è nota fin dall'antichità: divide et impera (dividi e regola)"<sup>2</sup>. Quando si progetta un sistema software complesso, è essenziale scomporlo in parti più piccole, ciascuna delle quali può essere perfezionata in modo indipendente.**

In questo modo, soddisfiamo il vincolo reale che esiste sulla capacità di canalizzazione della cognizione umana: per comprendere un determinato livello di un sistema, è sufficiente comprendere solo alcune parti (anziché tutte) contemporaneamente. Questo principio porta a una progettazione e a uno sviluppo modulare del software, favorendone i controlli di correttezza, la modularità e la riusabilità.

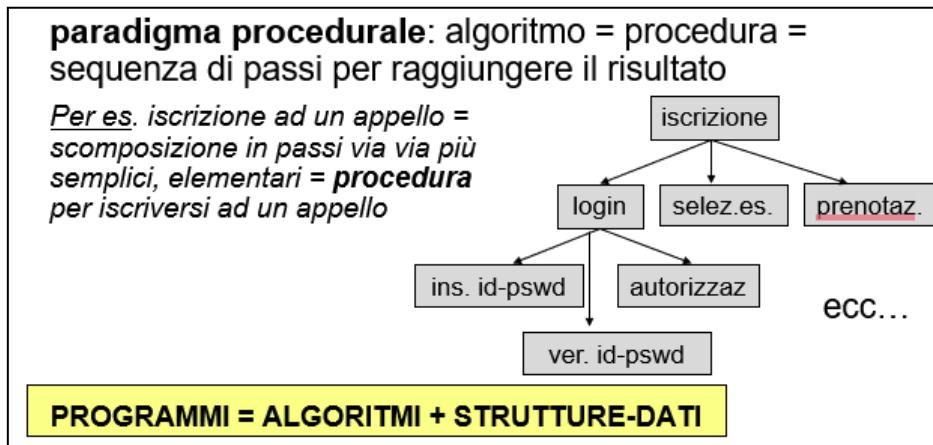
### Scomposizione algoritmica

La maggior parte di noi è stata formata con il paradigma della progettazione strutturata dall'alto verso il basso e quindi si approccia alla scomposizione dei problemi come a una semplice questione di scomposizione algoritmica, in cui ogni modulo del sistema denota una fase importante di un processo generale. Di seguito è riportato un esempio di uno dei prodotti della progettazione strutturata, un diagramma di struttura che mostra le relazioni tra i vari elementi funzionali della soluzione.

Il diagramma illustra parte della progettazione di un programma che supporta l'iscrizione degli studenti agli esami. La procedura comprende la fase di login (per supportare l'utente nella registrazione con le proprie credenziali - "login"), la selezione dell'esame a cui iscriversi ("selez. es.") e la fase di prenotazione ("prenotaz.") in cui lo studente si iscrive

<sup>2</sup> Dijkstra, E. 1979. Programming Considered as a Human Activity. In Classics in Software Engineering. New York, NY: Yourdon Press, p. 5.

all'esame. Queste fasi sono di per sé complesse e scomposte in algoritmi più semplici che mostriamo solo parzialmente nella figura.



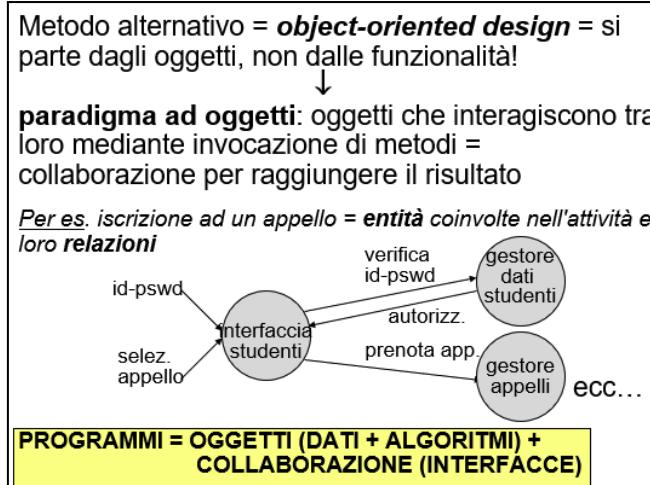
## Scomposizione Object-Oriented

Esiste una decomposizione alternativa per lo stesso problema. Di seguito, abbiamo **scomposto il sistema in base alle astrazioni-chiave del dominio del problema**. Anziché scomporre il problema in fasi come il login, la scelta dell'esame (selez. es.) e la prenotazione dell'esame (prenotaz.), abbiamo identificato oggetti che rappresentano servizi:

- l'interfaccia utente che lo studente utilizza per iscriversi all'esame ("interfaccia studenti");
  - l'oggetto che memorizza e gestisce i dati dello studente (per la verifica della password - "gestore dati studenti");
  - l'oggetto che memorizza i dati sugli esami (per recuperare le informazioni sugli esami disponibili, con le relative date - "gestore appelli").

Questi oggetti<sup>3</sup> rappresentano dei sotto-servizi dell'applicazione principale e derivano direttamente dal vocabolario del dominio del problema.

<sup>3</sup> In questo capitolo useremo il termine "oggetto" in modo rigoroso, ma ampio, per definire una parte software e/o un servizio. Nei capitoli successivi, lo stesso termine rappresenterà anche e soprattutto un costrutto linguistico della OOP.



Sebbene entrambi i progetti risolvano lo stesso problema, lo fanno in modi piuttosto diversi. **In questa seconda decomposizione, vediamo il mondo come un insieme di agenti autonomi che collaborano per eseguire un comportamento di livello superiore.** Per esempio, "verifica id-pswd" non esiste come algoritmo indipendente, ma è un'operazione associata all'oggetto "gestore dati studenti". In questo modo, **ogni oggetto della nostra soluzione incarna il suo comportamento unico e ognuno di essi modella qualche oggetto del mondo reale.**

Da questa prospettiva, **un oggetto è semplicemente un'entità tangibile che esibisce un comportamento ben definito. Gli oggetti fanno delle cose e noi chiediamo loro di eseguire ciò che fanno inviando loro dei messaggi.** Poiché la nostra scomposizione si basa sugli oggetti e non sugli algoritmi, la chiamiamo scomposizione orientata agli oggetti.

Si noti che il modello di intersazione nella scomposizione Object-Oriented si basa su un'**architettura client-server**:

- l'oggetto che offre uno o più servizi (metodi) è il server (per es., "gestore dati studenti");
- gli oggetti che invocano il server per ricevere i risultati dell'esecuzione del servizio sono i client (da ora in poi anche indicati come "consumatori" - consumer objects (per es., "interfaccia studenti").

Nell'esempio sopra, "Interfaccia studenti" è client sia di "gestore dati studenti" che di "gestore appelli".

#### **Confronto tra scomposizione algoritmica e Object-Oriented**

*Qual è il modo giusto di scomporre un sistema complesso - per algoritmi o per oggetti?*

Sono importanti entrambe le viste:

- La vista algoritmica mette in evidenza l'ordinamento degli eventi, mentre quella orientata agli oggetti enfatizza gli agenti che causano l'azione o sono i soggetti su cui agiscono le operazioni. Tuttavia, resta il fatto che non possiamo costruire un sistema complesso in entrambi i modi contemporaneamente, perché si tratta di viste completamente ortogonali. Dobbiamo iniziare a scomporre un sistema per algoritmi o per oggetti e poi usare la struttura risultante come quadro per esprimere l'altra prospettiva.
- La nostra esperienza ci porta ad applicare prima la vista orientata agli oggetti, perché questo approccio ci aiuta meglio a organizzare la complessità intrinseca dei sistemi software, così come ci ha aiutato a descrivere la complessità organizzata di sistemi complessi diversi come computer, piante e grandi istituzioni sociali. La scomposizione orientata agli oggetti produce sistemi più piccoli attraverso il riutilizzo di meccanismi comuni, fornendo così un'importante economia di espressione.

*Cosa significa riutilizzare i metodi (meccanismi) comuni?*

Si può notare che, nella scomposizione algoritmica di cui sopra, diverse fasi della procedura principale lavorano sugli stessi dati. Per esempio, entrambe le azioni "selez. es." e "prenotaz." interagiscono con lo stesso database per mostrare gli esami disponibili e inserire la scelta dello studente. Le procedure conoscono le strutture dati e accedono direttamente a tali dati per leggere/scrivere informazioni. Tuttavia, "selez. es." potrebbe essere riutilizzata in un'altra applicazione che supporta gli insegnanti nella modifica delle date degli esami. Quindi, sarebbe conveniente condividere questa procedura attraverso le applicazioni piuttosto che riscrivere programmi per svolgere compiti simili!

NOTA: rendendo esplicativi gli oggetti, la scomposizione object-oriented isola le funzionalità offerte dagli oggetti e i dati che essi gestiscono. Inoltre, gli oggetti possono proteggere i dati da accessi esterni, restituendo le informazioni necessarie o eseguendo operazioni invocate dagli oggetti consumatori. L'unica cosa che i consumatori devono sapere sono le operazioni offerte da questi oggetti (che in Java e nella maggior parte dei linguaggi ad oggetti sono dette *intefaccia pubblica del sistema*).

### **Il ruolo dell'astrazione**

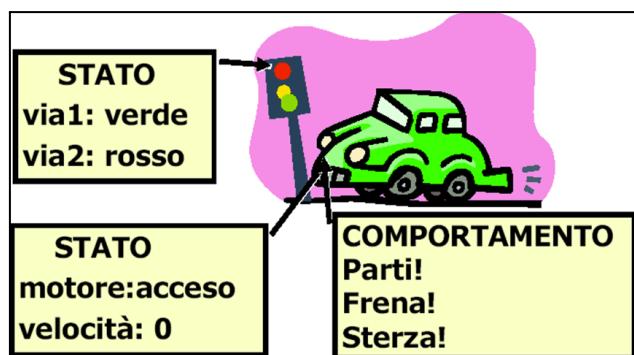
Miller osserva che: "L'arco del giudizio assoluto e l'arco della memoria immediata impongono gravi limitazioni alla quantità di informazioni che siamo in grado di ricevere, elaborare e ricordare. Organizzando l'input dello stimolo simultaneamente in

diverse dimensioni e successivamente in una sequenza di pezzi, riusciamo a rompere [ . . . ] questo collo di bottiglia informativo".<sup>4</sup> In termini contemporanei, chiamiamo questo processo **chunking o astrazione**.

Come lo descrive Wulf, "Noi (esseri umani) abbiamo sviluppato una tecnica eccezionalmente potente per affrontare la complessità. Astraiamo da essa. **Incapaci di padroneggiare la totalità di un oggetto complesso, scegliamo di ignorarne i dettagli inessenziali, occupandoci invece di un modello generalizzato e idealizzato dell'oggetto**".<sup>5</sup> Per esempio, quando si studia il funzionamento della fotosintesi in una pianta, ci si può concentrare sulle reazioni chimiche in alcune cellule di una foglia e ignorare tutte le altre parti, come le radici e gli steli.



Per fare un altro esempio, quando si modella il funzionamento di un semaforo, ci si può concentrare sul colore illuminato (verde, giallo o rosso) per capire l'istruzione che dà, senza preoccuparsi di come è implementato. Allo stesso modo, possiamo guidare un'automobile senza sapere come funzionano o sono strutturati il motore, gli ingranaggi, ecc. L'unica cosa importante da sapere è il **protocollo di interazione**, cioè come possiamo agire sui comandi (fisici) dell'auto per istruirla.



**La scomposizione orientata agli oggetti di cui sopra mostra esattamente questo tipo di astrazione: "gestore dati studenti" e**

<sup>4</sup> Miller. *Magical Number*, p. 95.

<sup>5</sup> Shaw, M., Feldman, G., Fitzgerald, R., Hilfinger, P., Kimura, I., London, R., Rosenberg, J., and Wulf, W. 1981. *Validating the Utility of Abstraction Techniques*. In *ALPHARD: Form and Content*, ed. M. Shaw. New York, NY: Springer-Verlag.

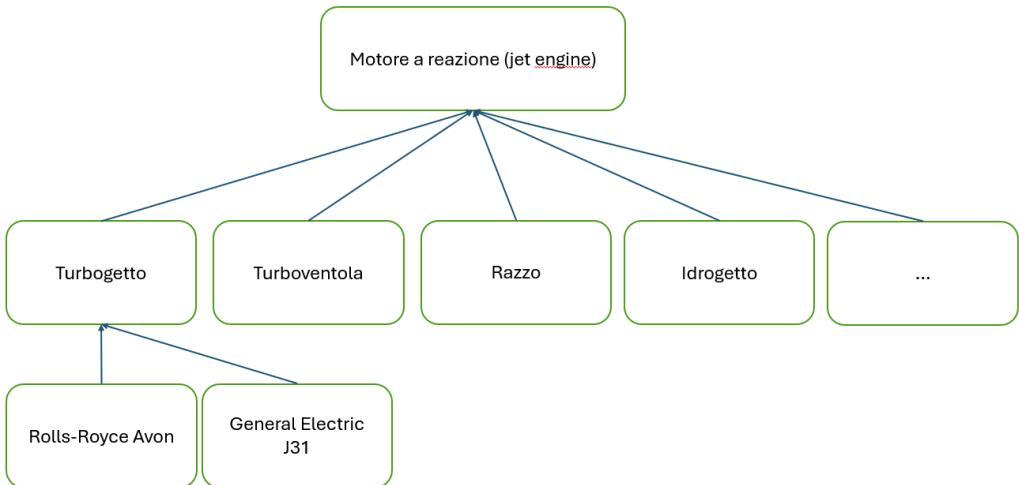
**"gestore appelli"** sono visti come astrazioni che forniscono servizi e non ci interessa come implementano i servizi stessi. Ad esempio, sfruttano un file per memorizzare i dati? O un database? Che altro? Non importa, basta che possiamo invocarli per recuperare/inserire dati (iscrizioni agli esami, ecc.) attraverso di essi. **Essendo agnostica rispetto all'implementazione, l'applicazione client (consumer) non subirà alcuna conseguenza se l'implementazione di "gestore dati studenti" e "gestore appelli" verrà modificata** (ad esempio, per migliorarne le prestazioni).

### Il ruolo della gerarchia (hierarchy)

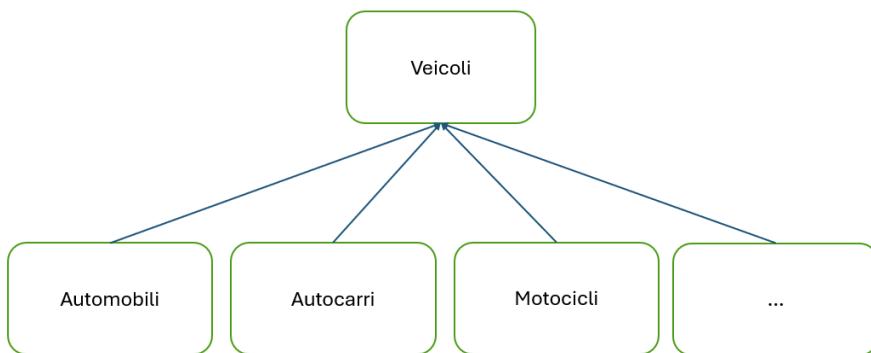
Un altro modo per aumentare il contenuto semantico dei singoli pezzi di informazione è quello di riconoscere esplicitamente le gerarchie esistenti all'interno di un sistema software complesso. La maggior parte dei sistemi interessanti non incarna un'unica gerarchia; al contrario, troviamo che all'interno dello stesso sistema complesso sono presenti molte gerarchie diverse. Ad esempio, un aereo può essere studiato scomponendolo in sistema di propulsione, sistema di controllo del volo e così via. Questa scomposizione rappresenta una **gerarchia strutturale o "part of"**.



Tuttavia, possiamo suddividere il sistema in modo del tutto ortogonale. Ad esempio, un motore a turbogetto è un tipo specifico di motore a reazione e un Rolls-Royce Avon è un tipo specifico di motore a turbogetto. In altre parole, un motore a reazione rappresenta una generalizzazione delle proprietà comuni a ogni tipo di motore a reazione; un motore a turbogetto è semplicemente un tipo specializzato di motore a reazione, con proprietà che lo distinguono, ad esempio, dai motori a turboventola. Questa seconda gerarchia rappresenta una **gerarchia (di ereditarietà) "is a"**.



Un altro esempio di gerarchia "is a": si pensi a "veicolo" come a un concetto che rappresenta le proprietà comuni a qualsiasi tipo di veicolo; poi, auto, camion, ecc. sono tipi specifici di veicoli ed ereditano le proprietà definite in "veicolo".



L'esperienza dei creatori del paradigma Object Oriented indica che **è essenziale vedere un sistema da entrambe le prospettive, studiando la sua gerarchia "is a" e la sua gerarchia "part of"**. Chiamiamo queste gerarchie rispettivamente **struttura delle classi e struttura degli oggetti del sistema**.

- **Struttura degli oggetti (gerarchia PART-OF)**: illustra come i diversi oggetti collaborano tra loro attraverso schemi di interazione che chiamiamo meccanismi.
- **Struttura delle classi (gerarchia IS-A)**: evidenzia la struttura e il comportamento comuni all'interno di un sistema.

Classificando gli oggetti in gruppi di astrazioni correlate (ad esempio, tipi di cellule vegetali rispetto a cellule animali), arriviamo a distinguere esplicitamente le proprietà comuni e distinte dei diversi oggetti, il che ci aiuta a padroneggiare la loro complessità intrinseca.

L'identificazione delle gerarchie all'interno di un sistema software complesso spesso non è facile perché richiede la scoperta di modelli tra molti oggetti, ognuno dei quali può incarnare un comportamento complicato. Una volta scoperte queste gerarchie, tuttavia, la struttura di un sistema complesso, e di conseguenza la sua comprensione, si semplifica notevolmente.

## Sommario

- **Il software è intrinsecamente complesso;** la complessità dei sistemi software spesso confonde la capacità intellettuale umana.
- Il compito del team di sviluppo del software è quello di creare l'illusione della semplicità.
- **La complessità assume spesso la forma di una gerarchia;** è utile modellare sia la gerarchia "is a" che la relazione "part of" di un sistema complesso.
- Esistono fattori limitanti fondamentali della cognizione umana; possiamo affrontare questi vincoli attraverso l'uso della **scomposizione, dell'astrazione e della gerarchia**.
- I sistemi complessi possono essere visti concentrandosi sulle cose o sui processi; ci sono ragioni convincenti per applicare la scomposizione orientata agli oggetti, in cui vediamo il mondo come una collezione significativa di oggetti che collaborano per raggiungere un comportamento di livello superiore.

Approfondiamo i temi principali che studieremo

### **Object-Oriented Programming**

La programmazione orientata agli oggetti è un **metodo di implementazione in cui i programmi sono organizzati come collezioni cooperative di oggetti**, ognuno dei quali rappresenta un'istanza di una classe e le cui classi sono tutte membri di una gerarchia di classi unite da relazioni di ereditarietà.

Questa definizione include tre importanti parti:

1. La programmazione orientata agli oggetti **utilizza come elementi logici fondamentali gli oggetti, non gli algoritmi** (gerarchia PART-OF);
2. **Ogni oggetto è un'istanza di una classe;**
3. **Le classi possono essere collegate tra loro tramite relazioni di ereditarietà (gerarchia IS-A).**

### **Object-Oriented Design**

I metodi di programmazione si concentrano principalmente sull'uso corretto ed efficace di particolari meccanismi linguistici. Al contrario, i metodi di progettazione pongono l'accento sulla strutturazione corretta ed efficace di un sistema complesso. Che cos'è dunque la progettazione orientata agli oggetti (OOD)?

Suggeriamo quanto segue:

- La progettazione orientata agli oggetti è un **metodo di progettazione che comprende il processo di scomposizione orientata agli oggetti e una notazione per rappresentare modelli logici** (struttura delle classi e degli oggetti) **e fisici** (architettura dei moduli e dei processi), **nonché statici e dinamici del sistema da progettare.**

Utilizzeremo il termine progettazione orientata agli oggetti per riferirci a qualsiasi metodo che porti a una scomposizione object-oriented.

### **Object-Oriented Analysis**

L'analisi orientata agli oggetti (OOA) enfatizza la costruzione di modelli del mondo reale, utilizzando una visione del mondo orientata agli oggetti:

- L'analisi orientata agli oggetti è un **metodo di analisi che esamina i requisiti dalla prospettiva delle classi e degli oggetti** presenti nel vocabolario del dominio del problema.

### **Che rapporto c'è tra OOA, OOD e OOP?**

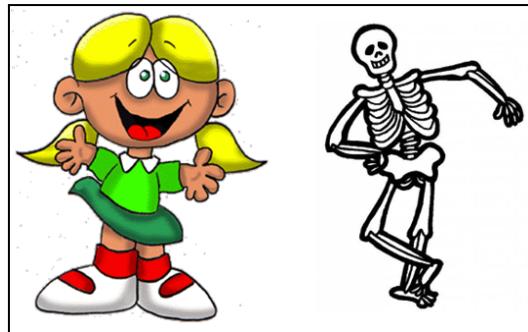
I prodotti dell'analisi orientata agli oggetti servono come modelli da cui partire per la progettazione orientata agli oggetti; i prodotti della progettazione orientata agli oggetti possono poi essere utilizzati come progetti (blueprint) per l'implementazione completa di un sistema utilizzando metodi di programmazione orientati agli oggetti.

## **Il significato dell'astrazione**

**Un'astrazione denota le caratteristiche essenziali di un oggetto che lo distinguono da tutti gli altri tipi di oggetti**, fornendo così confini concettuali ben definiti, rispetto alla prospettiva dell'osservatore.

**Un'astrazione si concentra sulla visione esterna di un oggetto** e serve quindi a separare il comportamento essenziale di un oggetto dalla sua implementazione.

*Un'astrazione rappresenta le caratteristiche essenziali e distintive di un oggetto, dal punto di vista di chi lo guarda.* La stessa entità può essere vista come una bambina da una persona qualunque, ma come l'interno del suo corpo (per esempio lo scheletro come in figura) da un medico.



Analogamente, una pianta può essere vista come insieme di steli, radici e foglie, o come struttura cellulare.



Possiamo caratterizzare il comportamento di un oggetto considerando i servizi che fornisce ad altri oggetti e le operazioni che può eseguire su altri oggetti. Questa visione ci costringe a concentrarci sulla vista esterna di un oggetto e ci porta a quello che Meyer chiama il **modello contrattuale della programmazione**:<sup>6</sup> la **vista esterna di ogni oggetto definisce un contratto da cui gli altri oggetti possono contare e che a sua volta deve essere eseguito dalla vista interna dell'oggetto stesso** (spesso in collaborazione con altri oggetti). Questo contratto stabilisce quindi tutte le ipotesi che un oggetto client può fare sul comportamento di un oggetto server. In altre parole, questo contratto comprende le responsabilità di un oggetto, cioè il comportamento per il quale è ritenuto responsabile (*accountable*).<sup>7</sup>

### Esempi di astrazione

Il primo esempio riguarda la definizione del comportamento di un **sensore di temperatura passivo**: un oggetto **client deve operare su un oggetto Sensore di temperatura per determinare la sua temperatura corrente**. Si pensi per esempio a un termometro corporeo. Il client può anche calibrare il sensore, ma noi trascuriamo questa funzione.

---

<sup>6</sup> Meyer, B. 1988. Object-Oriented Software Construction. New York, NY: Prentice Hall.

<sup>7</sup> Wirfs-Brock, R., and Wilkerson, B. October 1989. Object-Oriented Design: A Responsibility-Driven Approach. SIGPLAN Notices vol. 24(10).

|  |
|--|
| <b>Astrazione:</b> Sensore di temperatura                                |
| <b>Caratteristiche importanti:</b><br>temperatura                        |
| <b>Responsabilità:</b><br>riportare la temperatura corrente<br>calibrare |

Una diversa astrazione è un **sensore di temperatura "qualitativo**, che invece di fornire il valore esatto della temperatura rilevata ne restituisce un'interpretazione nella scala **{normale, alta}** (un po' come i sensori di temperatura corporea semplificati, che si colorano di verde se la temperatura è nel range normale e di rosso se c'è la febbre). Un client di questa astrazione può invocare un'operazione per stabilire un intervallo critico di temperature. In questo caso, quando un'applicazione invoca il sensore per conoscere la temperatura, il risultato sarà uno dei due valori previsti (normale / alta). Il sensore calcolerà quale valore restituire, applicando una funzione in base al valore reale di temperatura rilevato e al setpoint scelto (per esempio, il setpoint potrebbe essere 37 gradi C.).

|   |
|---|
| <b>Astrazione:</b> Sensore di temperatura qualitativo   |
| <b>Caratteristiche importanti:</b><br>temperatura<br>setpoint                                     |
| <b>Responsabilità:</b><br>riportare la temperatura corrente<br>calibrare<br>stabilire il setpoint |

Il modo in cui il sensore di temperatura attivo adempie alle sue responsabilità è una funzione interna e non riguarda i clienti esterni. Questi sono quindi i **segreti della classe (secrets)**, che vengono implementati dalle parti private della classe insieme alla definizione delle sue funzioni membro.

## Il significato di encapsulamento (encapsulation, information hiding)

Qualsiasi rappresentazione venga scelta è irrilevante per il contratto tra il client e il sensore di temperatura qualitativo,

purché tale rappresentazione rispetti il contratto. In poche parole, **l'astrazione di un oggetto dovrebbe precedere le decisioni sulla sua implementazione**. Una volta scelta l'implementazione, questa dovrebbe essere trattata come un segreto dell'astrazione e nascosta alla maggior parte dei clienti.

**L'astrazione e l'incapsulamento sono concetti complementari:**

- **L'astrazione si concentra sul comportamento osservabile di un oggetto;**
- **L'incapsulamento si concentra sull'implementazione** che dà origine a questo comportamento.

L'incapsulamento si ottiene più spesso attraverso l'information hiding (non solo il data hiding), ovvero il processo di **nascondere tutti i segreti di un oggetto che non contribuiscono alle sue caratteristiche essenziali**; in genere, **la struttura di un oggetto viene nascosta, così come l'implementazione dei suoi metodi**.

"**Nessuna parte di un sistema complesso dovrebbe dipendere dai dettagli interni di un'altra parte**". Mentre l'astrazione "aiuta le persone a pensare a ciò che stanno facendo", l'incapsulamento "consente di apportare modifiche al programma in modo affidabile con uno sforzo limitato".<sup>8</sup>

• **Incapsulamento (information hiding)**



L'incapsulamento (o *information hiding*) è il principio secondo cui la struttura interna, il funzionamento interno, di un oggetto **non deve essere visibile** dall'esterno

Questo significa che **ogni classe deve avere due parti**:

**un'interfaccia e un'implementazione**. L'interfaccia di una classe cattura solo la sua visione esterna, comprendendo la nostra astrazione del comportamento comune a tutte le istanze della classe. L'implementazione di una classe comprende la rappresentazione dell'astrazione e i meccanismi che consentono di ottenere il comportamento desiderato. L'interfaccia di una classe è l'unico luogo in cui si affermano tutte le ipotesi che un client

<sup>8</sup> Gannon, J., Hamlet, R., and Mills, H. July 1987. Theory of Modules. IEEE Transactions on Software Engineering vol. SE-13(7), p. 820.

può fare su qualsiasi istanza della classe; l'implementazione incapsula i dettagli su cui nessun client può fare ipotesi.

### Esempio di incapsulamento

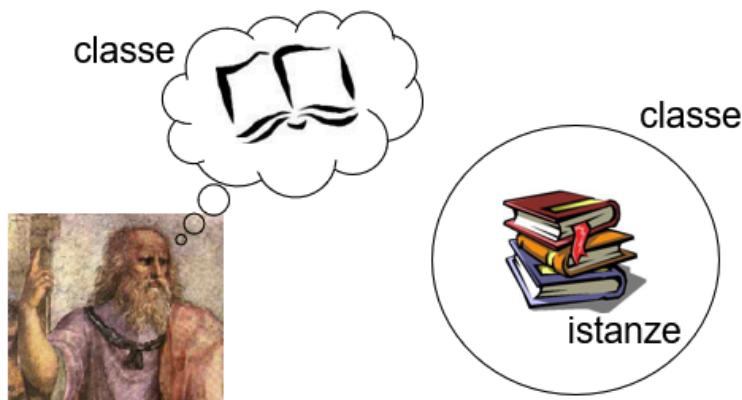
Uno scaldino (heater) si trova a un livello di astrazione piuttosto basso e quindi potremmo decidere che ci sono solo tre operazioni significative che possiamo eseguire su questo oggetto: accenderlo, spegnerlo e scoprire se è in funzione. Tutto ciò che un client deve sapere sulla classe Heater è la sua interfaccia disponibile (cioè le responsabilità che può eseguire su richiesta del client).

L'incapsulamento intelligente localizza le decisioni di progettazione che probabilmente cambieranno. Con l'evoluzione di un sistema, gli sviluppatori potrebbero scoprire che, nell'uso reale, alcune operazioni richiedono più tempo di quanto sia accettabile o che alcuni oggetti consumano più spazio di quello disponibile. In queste situazioni, la rappresentazione di un oggetto viene spesso modificata in modo da poter applicare algoritmi più efficienti o da ottimizzare lo spazio calcolando anziché memorizzando determinati dati. **L'incapsulamento permette di cambiare la rappresentazione di un'astrazione senza impattare sui suoi client.**

### Classi e oggetti (istanze)

#### Un'istanza (oggetto) è...

- Un singolo libro concreto (che può essere preso in prestito, restituito, distrutto, fotocopiato, ecc...)

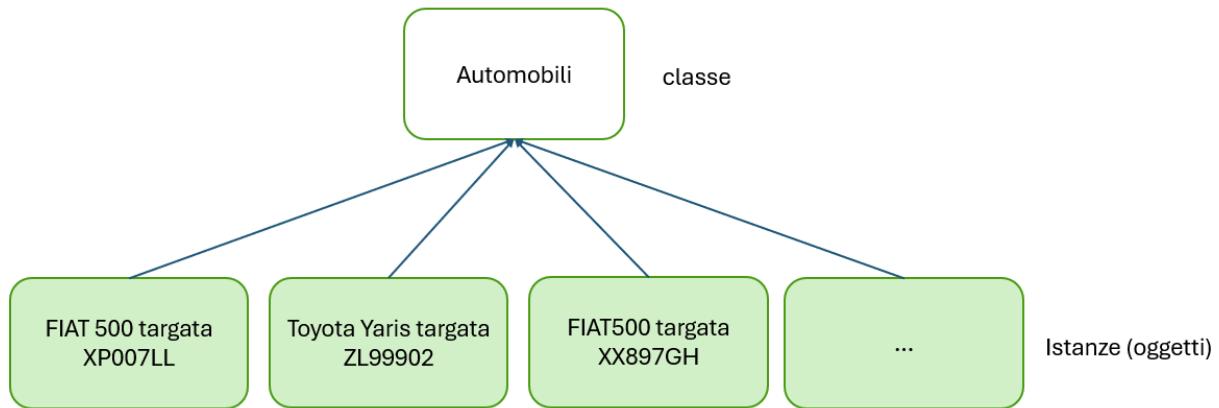


**Un oggetto (o istanza) è un'entità che ha uno stato, un comportamento e un'identità. La struttura e il comportamento di oggetti simili sono definiti nella loro classe comune.**

Anche se trattiamo ogni istanza di un particolare tipo di oggetto come distinta, possiamo assumere che essa condivida lo stesso comportamento di tutte le altre istanze dello stesso tipo di oggetto. Indichiamo il tipo di oggetto come "classe": una classe è un insieme di oggetti che condividono una struttura comune, un comportamento comune e una semantica comune. Una classe rappresenta l'insieme degli oggetti che condividono le proprietà che essa specifica. Diciamo anche che tali oggetti ereditano le proprietà definite dalla classe.

Se la classe "auto" specifica che le auto hanno una targa e 4 ruote, allora tutte le sue istanze (la mia Toyota Yaris, la Nuova FIAT 500 di mio fratello, ...) ereditano le proprietà; per esempio, assumiamo che non ci siano istanze di auto con 3 ruote o con più di 4.

**Il potere dell'ereditarietà è la sintesi:** definiamo le proprietà nelle classi, che sono relativamente limitate in numero, e le loro numerose istanze le ereditano senza richiedere una descrizione individuale delle proprietà dell'oggetto. Questo aiuta la comprensione (nessun sovraccarico di informazioni) e lo sviluppo/manutenzione del software.



### **Stato di un oggetto**

**Lo stato di un oggetto include i valori delle sue proprietà.** Per esempio: la data di nascita di una persona e il peso.



**Tutte le proprietà hanno un valore, che può essere una semplice quantità, o può denotare un altro oggetto.** Per esempio:

- Una parte dello stato di un ascensore potrebbe mantenere il valore 3, che denota il piano in cui si trova l'ascensore in un certo istante.
- Lo stato di un distributore automatico può includere diversi oggetti, come un insieme di bibite. Ogni bibita è un oggetto distinto e le sue proprietà sono diverse da quelle del distributore (le bibite possono essere bevute, il distributore no); inoltre, si opera sulle proprietà delle bibite in modo diverso da come si opera su quelle del distributore (la bibita viene consumata, il distributore viene utilizzato per acquistarla).



**Altro esempio: la classe Impiegato e alcune sue istanze.**

Definizione della classe Impiegato con i suoi attributi principali:

| Impiegato   |
|---|
| - Nome:<br>- Codice Fiscale:<br>- Dipartimento:<br>- Salario: |

Ed ecco due istanze di Impiegato:

| Impiegato   |
|---|
| - Nome: Mario Rossi<br>- Codice Fiscale: RSSMR...<br>- Dipartimento: Informatica<br>- Salario: 2400 |

| Impiegato  |
|--|
| - Nome: Giorgia Neri<br>- Codice Fiscale: NREGRG...<br>- Dipartimento: Matematica<br>- Salario: 2500 |

**È buona pratica di Software Engineering incapsulare lo stato di un oggetto anziché esporlo pubblicamente: l'esposizione pubblica permetterebbe ad applicazioni esterne di modificare lo stato, liberamente. Per incapsulare, potremmo modificare l'astrazione (la classe) aggiungendo le operazioni (responsibilities) che offre:**

| Impiegato  |
|--|
| - Nome:<br>- Codice Fiscale:<br>- Dipartimento:<br>- Salario:<br><br>+ Get Nome()<br>+ Get CF()<br>+ Get Dipartimento()<br>+ Get Salario() |

**Le "Responsibilities" di un oggetto sono tutti i servizi che offre (in termini di metodi offerti ai clienti) per tutti i contratti che**

**supporta (cioé, per tutti i ruoli che può giocare nel sistema complessivo).**

## Lezione 02

### Classi, attributi e metodi pubblici e privati

In questa lezione introduciamo in modo preciso la definizione di classe in Java e le sue componenti principali.

**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. Un'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 degenere (o oggetto "che non esiste") di indirizzo **null**, che non ha né campi, né metodi, lo stesso per tutte le classi. In Java un esempio di oggetto non banale è un array di interi. In particolare, un array di interi viene identificato con il suo indirizzo e consiste di un gruppo di interi disposti in modo consecutivo, insieme all'intero **length** che ne rappresenta la lunghezza).

**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 scegliamo 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.raza*, *x.anni*, e più in generale con *x.attributo*.

**Creazione di istanze.** Gli oggetti di una classe si costruiscono con un comando **new** che istanzia (ovvero costruisce) un nuovo oggetto come l'indirizzo di un'area di memoria. **Tale area conterrà gli attributi dell'oggetto.** Il comando `C x = new C();` crea un nuovo oggetto `x` nella classe `C`, con **valori di default** per gli attributi. La variabile `x`, a sua volta, contiene l'indirizzo dell'oggetto di classe `C` appena creato.

**IMPORTANTE!** Tutti gli oggetti vengono memorizzati in una zona di memoria detta **heap** e il loro indirizzo viene salvato nella variabile a cui si assegna il risultato di una `new()` (vedere sopra l'esempio `C x = new C();`). In Java, come in C, esiste anche la zona di memoria detta **stack**, che contiene i frame (record di attivazione) di esecuzione delle procedure e funzioni (dette **metodi** in Java), similmente a quanto succede in C. C'è una terza area di memoria, che si chiama area delle classi, che contiene varie informazioni sulle classi, in particolare lì sono memorizzate le **variabili statiche**, di cui parleremo in questo insegnamento.

**Eseguibilità di una classe.** Una classe è eseguibile solo se contiene il metodo `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 solo essere 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 PPOO**, 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 con indirizzo **null**, 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.automobilePosseduta;
System.out.println(auto.colore)
```

Quando eseguirò riceverò:

```
> exception... NullPointerException9
```

**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

<sup>9</sup> Controllate sempre se un valore ritornato è null: le Null Pointer Exceptions sono il più comune errore nei programmi Java e sono un autentico incubo da trovare perché non accadono a tempo di 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 NullPointerException che causano 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.

oggetti o il main programme) vedano come questi sono 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. Come specificato nella Lezione 01, il vantaggio dell'incapsulamento è che 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 un'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.

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.** Una classe in Java può contenere varie componenti. In laboratorio cominciate a vedere le componenti dette metodi **statici**, i quali possono aiutare a fare un parallelo con le funzioni del C. Un metodo statico prende dei valori in ingresso e restituisce un valore in uscita oppure modifica la memoria ed è simile a una funzione C. Una classe può anche contenere metodi **dinamici**<sup>10</sup>, che sono componenti propriamente object-oriented (capirete bene il perché man mano che proseguiremo nell'insegnamento). Un metodo dinamico prende dei valori in ingresso e restituisce un valore in uscita oppure modifica la memoria, ma rispetto a un metodo statico viene invocato **"mandando un messaggio"** a un oggetto, e ci restituisce **"la risposta dell'oggetto"**. Vedremo che in Java è possibile stabilire delle regole di encapsulamento, tramite parole-chiave detti *modificatori di visibilità*. All'inizio parleremo di metodi **pubblici** (utilizzabili fuori dalla classe in cui sono definiti) e **privati** (utilizzabili solo nella classe in cui sono definiti). Cominciamo spiegando i metodi **dinamici pubblici** (si noti che anche ai metodi statici si

---

<sup>10</sup> La terminologia Java corretta è metodi **di istanza**, non metodi dinamici, ma questa locuzione informale viene spesso usata per contrapporli in modo più diretto ai metodi statici.

possono associare dei modificatori di visibilità). 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 invocato mandando un messaggio a un oggetto. Questo determina l'esecuzione del metodo sull'oggetto x da parte della macchina virtuale Java. Sia x un oggetto di classe (quindi tipo) Gatto, se scrivo **x.metodo()** mando il messaggio "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 del metodo, come avviene di solito.

**Nota avanzata.** 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 su cui ho invocato 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**: in una variabile statica, 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. Le variabili statiche rappresentano un valore unico (non necessariamente costante - le costanti

si indicano con il modificatore "final") durante la computazione.

- Invece i nomi degli attributi vengono detti **variabili di istanza o 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, le variabili dinamiche hanno **un valore specifico 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.

**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 invocato "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 "nell'automobile, suona il clacson".

Gli attributi degli oggetti sono di regola dinamici, lasciando solo come statiche le caratteristiche che sono attribuibili alla classe di per sé 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). Vedrete meglio nelle prossime settimane, in laboratorio e in aula.

**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, í, ò, ù, È. Per la stessa ragione scriviamo '' e "" al posto di '' e di "", che alcuni compilatori non leggono.

Per completezza riportiamo un link a un sito web che presenta i **tipi primitivi di Java**:  
<https://www.html.it/pag/15101/tipi-primitivi-di-java-e-valori/>

Ora torniamo all'esempio della classe Gatto, presentandone il codice, completo di una classe di test con metodo main().

Nel codice che segue, useremo anche **esempi di metodi dinamici presi da una libreria Java**. Utilizziamo infatti una classe **Scanner**<sup>11</sup>, che offre metodi per fare l'input da tastiera. Uno oggetto di tipo 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() all'oggetto x di tipo Scanner, e se inseriamo una stringa da tastiera otteniamo in risposta la rappresentazione Java della stringa. Notate come nel codice dichiariamo la variabile di tipo Scanner come static: il motivo è che, all'interno del programma, ci serve un solo oggetto Scanner per leggere da tastiera.

```
//Salveremo il tutto nel file: GattoDemo.java
import java.util.Scanner; //Usiamo la classe Scanner (Java utility)
/* Nota: l'unica libreria Java che viene importata automaticamente è la java.lang12, a cui appartiene, per esempio il metodo System.out.println() che adoperiamo per stampare a video. */

class Gatto { /* Le classi iniziano con la maiuscola */
```

---

<sup>11</sup> <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>

<sup>12</sup> <https://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html>

```

// 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(){... ...}
Secondo una concezione ben stabilita, 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;
    }
}

/** Segue 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:

```

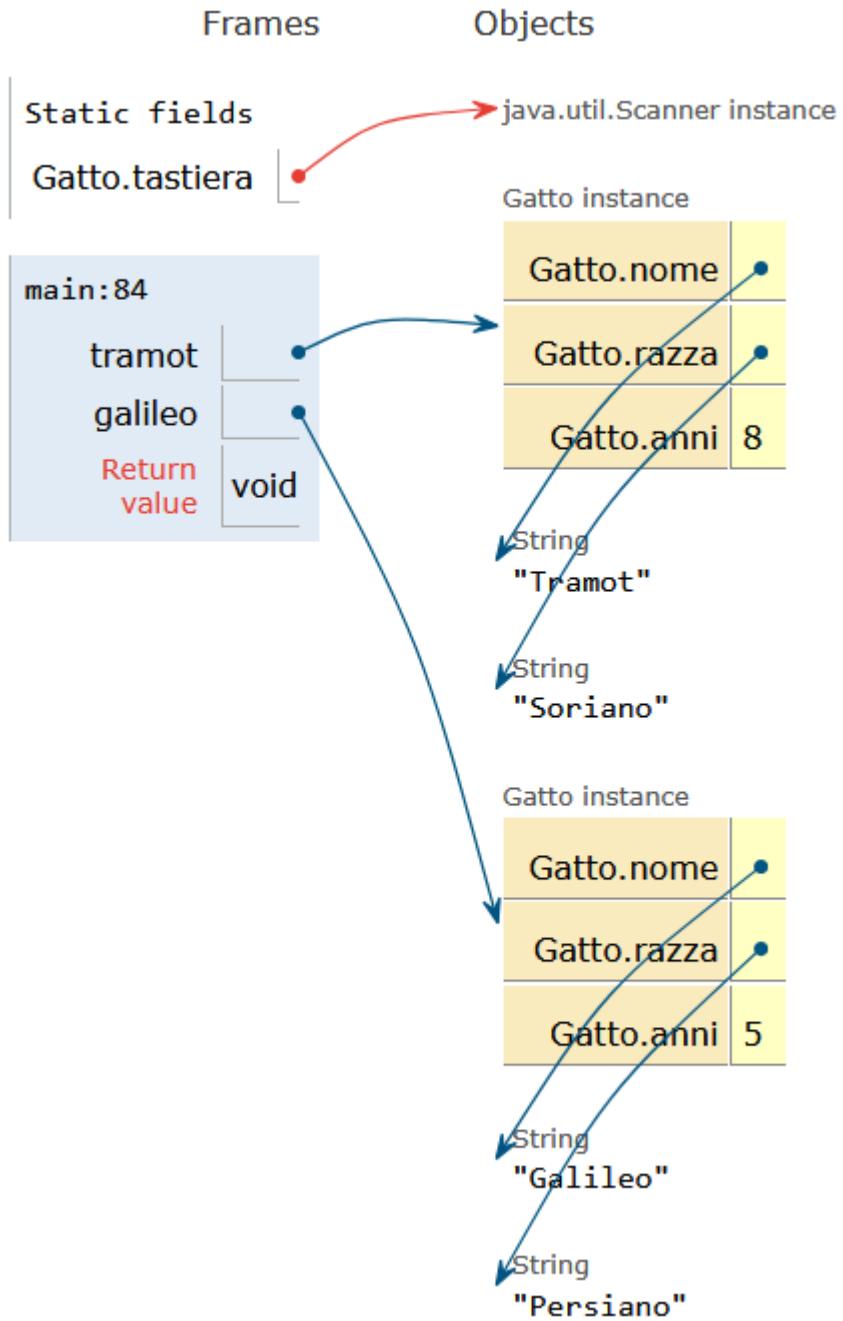
tramot prima inserimento dati
nome = null
razza = null
anni = 0
Inserisci dati tramot
nome = Tramot
razza = Soriano
anni = 8
Dati inseriti tramot
nome = Tramot
razza = Soriano
anni = 8
eta' tramot in anni umani 52
Inserisco i dati galileo
nome = Galileo
razza = Persiano
anni = 5 */

```

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

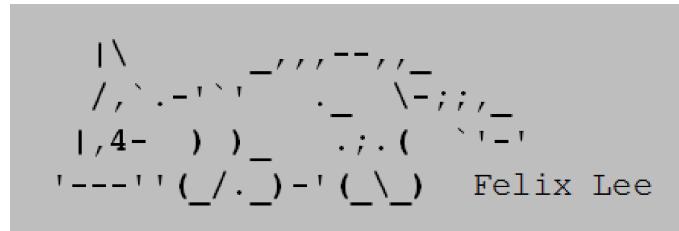
[https://cscircles.cemc.uwaterloo.ca/java\\_visualize/](https://cscircles.cemc.uwaterloo.ca/java_visualize/)

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 (cioè quelle variabili contengono degli indirizzi alla parte di memoria chiamata **heap**). 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 nel metodo leggiInput() il suo 'this' (il parametro implicito) è un alias di 'tramot', su cui chiamiamo il metodo stesso.

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

**Nota tecnica sul metodo `toString()`.** Il metodo dinamico **`String toString()`** con tipo di ritorno String e nessun parametro esplicito (ovvero nessun parametro tra parentesi) appartiene allo standard Java e in particolare è presente nella classe 'Object', genitore di tutte le classi, e di conseguenza in tutte le classi Java. Di default stampa in esadecimale l'indirizzo dell'oggetto. Il `toString()` è un metodo dinamico, quindi viene invocato su un oggetto e contiene la variabile隐式的 'this': se x è un oggetto di una classe C, allora, se scriviamo **`x.toString()`**, otteniamo la stringa di caratteri: "**C@indirizzo\_esadecimale\_x**". Questa stringa identifica univocamente x, e contiene informazioni come classe e indirizzo di memoria in esadecimale.

Il `toString()` può tuttavia venire sovrascritto (ridefinito) dal programmatore per costruire una stringa diversa, per esempio per fare la "pretty printing" dei valori dei campi degli oggetti: è così che succede nella classe 'Gatto'. Parleremo diffusamente 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 programmatori. Per esempio, quando passiamo un oggetto al metodo di stampa `System.out.println()`, come in:

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

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

Se la classe dell'oggetto non ha una sua versione di `toString()`, per esempio `x` di classe `C` senza `toString()`, con `System.out.println(x);` si ottiene la stampa di `x.toString()` generica (quella di `Object`), ovvero, come detto sopra, `"C@indirizzo_esadecimale_x"`.

Se la classe dell'oggetto, in questo caso 'Gatto', contiene una sua versione di `toString()`, allora verrà eseguita questa versione.

La `println()` semplicemente 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()!!!
*/
```

### **Nota sulla differenza tra struct di C e class di Java**

Avete incontrato il costrutto **struct** in C. Una domanda naturale che possiamo farci è: qual è la differenza tra le **struct** e le **classi**?

Scriviamo una **struct** come esempio:

```
/* Esempio di struct */

struct Gatto {

    char nome[];

    char razza[];

    int eta;

};
```

Le **struct** rappresentano tipi di dati **eterogenei**, cioè descrivono strutture che contengono una o più variabili di tipo diverso, a cui accedere in lettura o scrittura con la notazione **".":**

```
// uso della struct C

Gatto mioGatto; // variabile di tipo Gatto

mioGatto.nome = "Lilin";
```

Anche le **classi** rappresentano tipi di dati **eterogenei**, ma oltre ai dati (detti **attributi** o **campi**) contengono i metodi, ovvero operazioni che attuano il comportamento degli oggetti *istanze* delle classi, leggendone e/o modificandone i dati (detto *stato degli oggetti*). Inoltre gli oggetti sono allocati esplicitamente in memoria con una istruzione **new**, che, appunto, *crea le istanze della classe*:

```
// uso della class Java

Gatto mioGatto; // variabile di tipo Gatto

mioGatto = new Gatto(); // istanza!

mioGatto.nome = "Lilin";
```

Gli oggetti sono, quindi, indirizzi di una parte di memoria (strutturata) secondo la descrizione dello stato. L'indirizzo rappresenta la loro identità, per cui è possibile distinguere due oggetti diversi con lo stesso stato (stessi dati), invece due struct con lo stesso stato non sono distinguibili.

Inoltre, i dati in una struct possono essere letti e/o modificati da qualsiasi codice client. Invece lo stato di un oggetto può essere direttamente accessibile da qualsiasi codice oppure no. Questo implica la possibilità di *incapsularlo*, ovvero nasconderlo, al codice client, e quindi di cambiare i tipi dei suoi campi e l'implementazione dei metodi in modo trasparente al codice client stesso, che non dovrà essere riscritto.

Tutti questi concetti verranno ripresi e approfonditi in questo insegnamento, a partire dalla prossima lezione.

**Passaggio per valore o passaggio per riferimento?** In C avete visto due tecniche esplicite di passaggio dei parametri, **per valore** (viene passato il contenuto della variabile che è parametro attuale al parametro formale della funzione) e **per riferimento o referenza** (viene passato l'indirizzo della variabile parametro attuale al parametro formale della funzione). In Java esiste solo il passaggio **per valore**, ma attenzione: abbiamo detto che nelle variabili di programma che rappresentano oggetti viene memorizzato un indirizzo di memoria. Passare un oggetto come parametro in Java, vuol dire quindi passare il suo indirizzo, realizzando quindi una forma di passaggio per riferimento implicito.

**Nota su compilazione e esecuzione di Java.** In C, quando un programma sorgente viene compilato si ottiene un programma eseguibile il cui formato dipende dalla macchina/sistema operativo sulla quale si compila. Esiste quindi un compilatore per ogni macchina/sistema operativo. In Java, i programmi sorgenti vengono compilati (con comando 'javac nomefile.java') in un formato intermedio, detto **bytecode**, uguale per tutte le macchine/sistemi operativi. Il programma in formato intermedio

verrà quindi eseguito invocando l'interprete/macchina astratta di Java, chiamato Java Virtual Machine - JVM (con comando 'java nome file'). Il vantaggio di avere un formato intermedio generale e indipendente è quello di poter avere codice **portabile** sulle diverse macchine/sistemi operativi, lo svantaggio è che l'esecuzione dell'eseguibile Java in bytecode è un po' più lenta di un eseguibile nativo.

## Lezione 03

### Attributi e metodi privati, get e set

**Lezione 03. Parte 1. Un primo esempio di attributi e metodi privati: la classe Specie.**

Prima di tutto, ricordiamo che si tende a usare i termini "attributo", "campo" e "stato" di un oggetto in modo intercambiabile, anche se, rispettivamente, "attributo" si riferisce più alle caratteristiche astratte di un'entità in fase di *progettazione*, "campo" è la sua controparte in un'*implementazione* e "stato" è l'*insieme dei valori assunti* da tutti i campi a ogni passo di computazione.

Spesso è conveniente o necessario **rendere privati gli attributi di una classe** (per esempio, per impedire ad altri classi di lettere/scrivere direttamente tali attributi). **Un attributo privato può essere letto o modificato direttamente solo dall'interno della classe che lo definisce:** è come se fosse chiuso dentro una scatola che non si può aprire. Se si vuole offrire un accesso "controllato" agli attributi privati, bisogna che la classe fornisca dei metodi per accedere a tali attributi attraverso gli strumenti offerti dalla classe stessa. 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). Un grande 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.

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 ma con indirizzo (identità) diversa.

```
import java.util.Scanner;

/** Specie e' una classe non eseguibile per rappresentare
delle specie di esseri viventi. Scriveremo un programma per
sperimentare Specie in una classe di nome SpecieDemo, e
salveremo tutto nel file: SpecieDemo.java */
class Specie {
    /** Classe non pubblica, 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'
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();}

    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);
}

}

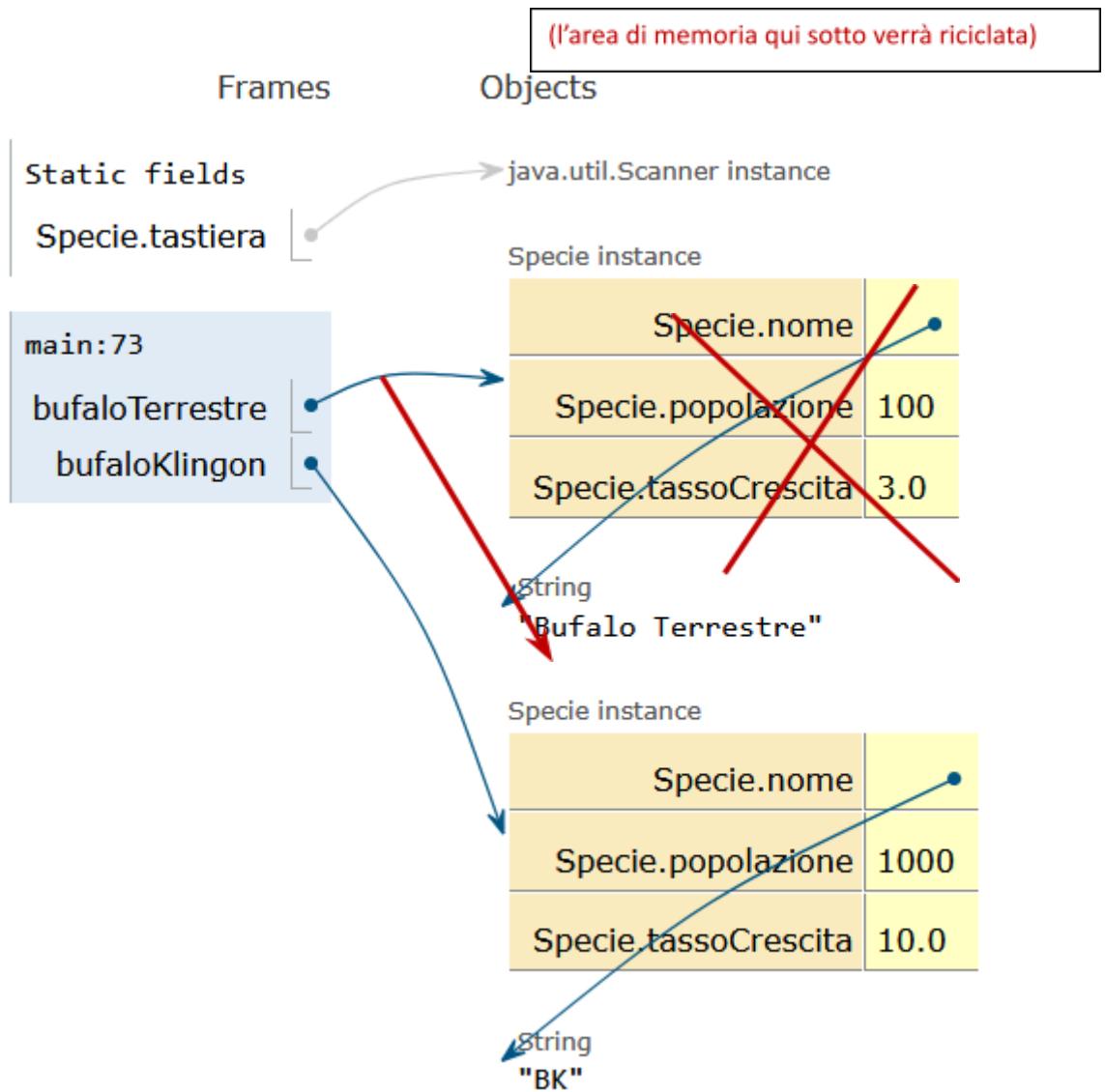
/** Effetto dell'assegnazione
bufaloTerrestre = bufaloKlingon;

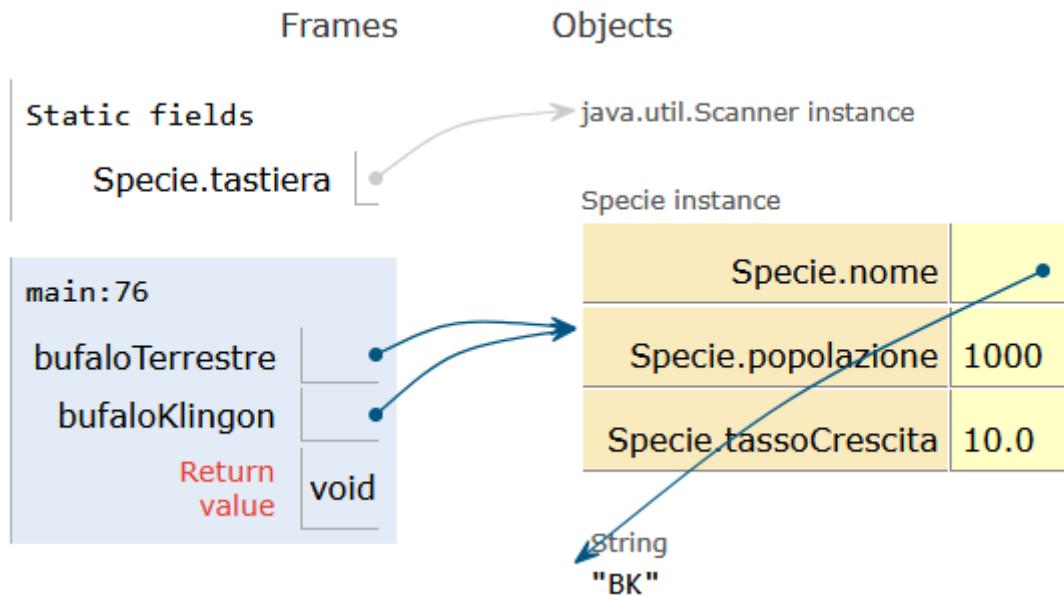
```

1. L'indirizzo di bufaloTerrestre diventa uguale a quello di bufaloKlingon. In altre parole, bufaloTerrestre adesso e' un alias di bufaloKlingon.
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/](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 `SpecieDemo`.** 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 **ricicljata**.

### Esercizio da fare a casa

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 in queste dispense nella Lezione 04.

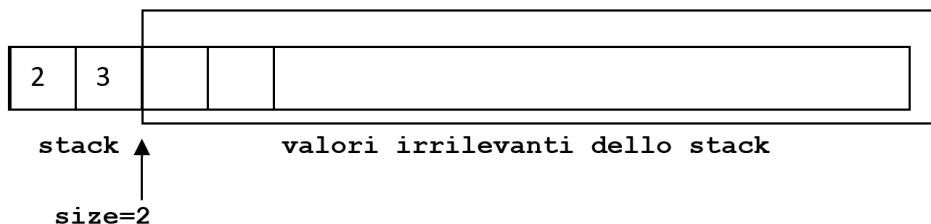
### Lezione 03. Parte 2. Un esempio non banale di attributi e metodi privati: la classe **Calcolatrice**. (50 minuti).

Vediamo ora un esempio non banale di classe che definisce attributi privati, ma anche metodi privati: la classe **Calcolatrice**. Gli elementi della classe sono oggetti di tipo **Calcolatrice**, ovvero "calcolatrici" tutte uguali, semplici robot virtuali che ricevono una lista di comandi da una tastiera e eseguono dei calcoli.

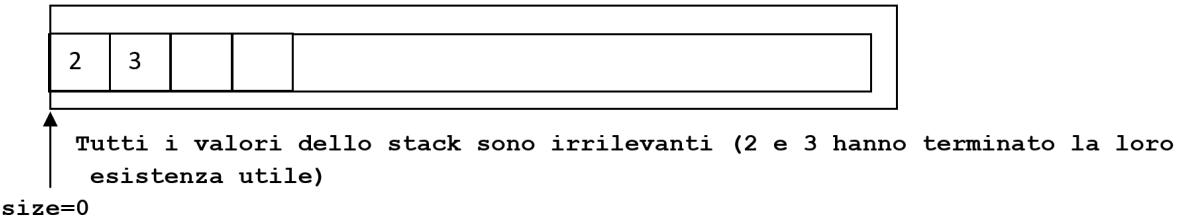
**Una calcolatrice semplificata.** Consideriamo solo operazioni tra interi (quindi niente tasto "virgola") e argomenti di positivi e di **una cifra** (quindi non dobbiamo preoccuparci di raccogliere le cifre per formare numeri più grandi). Abbiamo quindi le cifre '0' ... '9'. Come uniche operazioni consideriamo + 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 da utilizzare come struttura-dati **stack** (Last-In-First-Out; LIFO) 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, 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 restituiscia l'ultimo 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.



Ora la calcolatrice moltiplica 2 e 3 e ottiene 6.

**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 codice client utilizzatore della calcolatrice ha bisogno di conoscere è **esegui()**: tutti gli attributi e gli altri metodi (in particolare, quelli che gestiscono l'array come uno stack) riguardano il funzionamento interno degli oggetti "calcolatrice" e quindi è prudente vietarne l'uso. Renderemo quindi **esegui** **pubblico** e tutti gli attributi e gli altri metodi **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.

Vediamo ora perché abbiamo deciso di avere uno stack di supporto. Scegliamo 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 (**notazione postfissa**, anziché infissa). 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, anche se è più 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 interessante avere.

Segue il codice. È importante osservare che nel metodo esegui() non si vede che la struttura della calcolatrice è un array, poiché la si usa solo tramite le operazioni push() e pop(): se si cambiano push() e pop() esegui() non cambia. Questo è quindi un primo esempio d'uso dell'incapsulamento dei dati.

Fate qualche prova di esecuzioni di istruzioni con il Java Visualizer. Il programma che trovate nel main() di 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+"
```

```

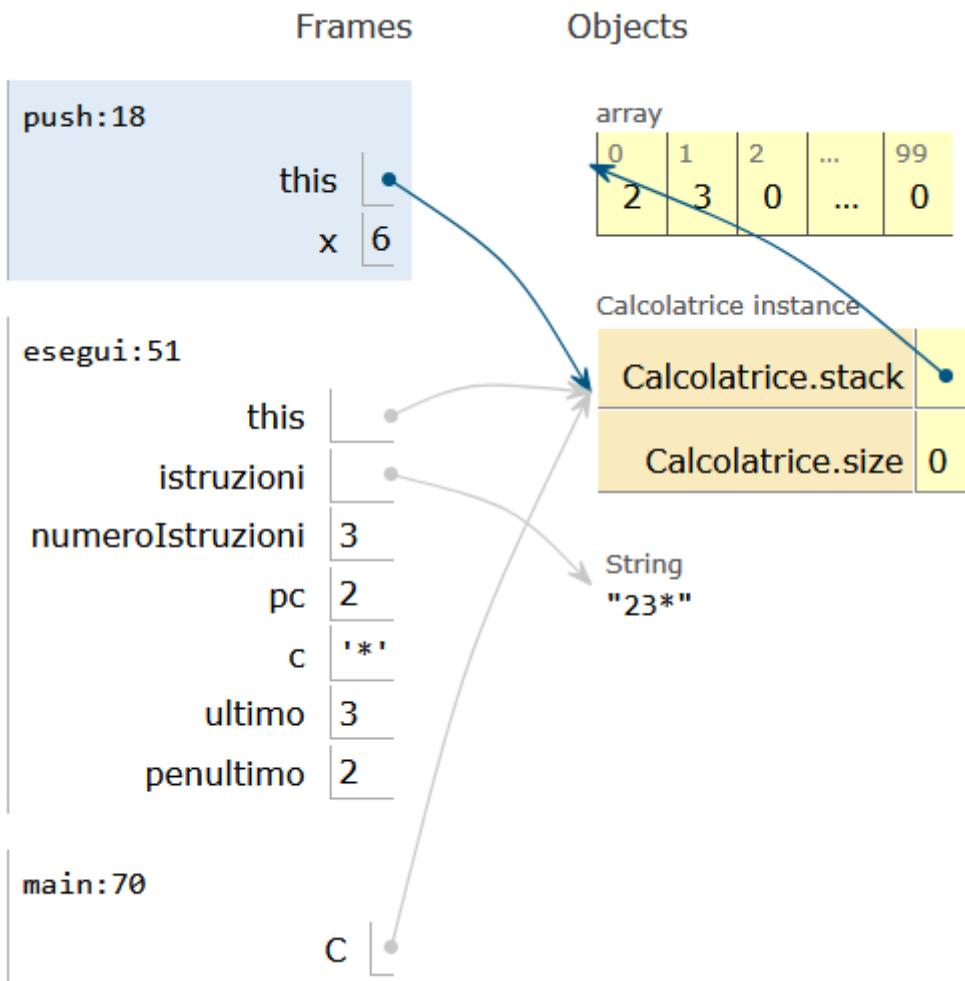
(i) La prima cifra viene inserita e lo stack passa da: {}
(vuoto) a
    {2} (contiene il solo 2).
(ii) La seconda cifra viene inserita e lo stack passa da {2}
a {2,3}
(iii) Quando leggiamo il + togliamo gli ultime due interi
dallo stack
    che ritorna ad essere vuoto: {}
(iv) Sommiamo i due interi: 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)**. Il 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("Eseguo 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 ***ArrayOutOfBoundsException***, prodotto dalla ricerca del secondo argomento per la prima somma, argomento che viene cercato in posizione -1.

## Lezione 04

### Assegnazioni di oggetti, metodo equals, costruttori

**Lezione 04. Parte 1.** Abbiamo visto che, data una classe C, scrivendo new C() costruiamo un nuovo oggetto nella classe C, con tutti gli attributi inizializzati con valori di default (ricordiamo che 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 gli 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 possiamo assegnare dei valori agli attributi della classe, utilizzando gli argomenti del metodo: per esempio, questi possono essere gli attributi del nuovo oggetto che stiamo costruendo. Si noti che 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 dal compilatore. 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 e 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` e `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**. Se chiamiamo su 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. Abbiamo messo "uguale" tra virgolette perché una versione di default del metodo `equals()` è a disposizione in ogni classe Java, ereditato dalla classe `Object` (abbiamo accennato a questa classe antenata di tutte le classi, quindi radice della gerarchia delle classi, quando abbiamo parlato di `toString()` nella Lezione 2). In questa versione di default, `x.equals(y)` effettua il controllo `x==y` (ovvero confronta i due indirizzi). Tuttavia, **noi possiamo ridefinire equals()** in ogni classe che implementiamo e quindi decidere quando consideriamo **due oggetti di una classe C uguali**: in genere si controlla che tutti gli attributi siano uguali, ma questa non è l'unica possibilità, visto che il concetto di "uguale" dipende da cosa rappresentano gli oggetti di una certa classe, è quindi parte della progettazione di una classe.

Vediamo un esempio, 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 e s' sono
gli oggetti legati rispettivamente a this (ovvero l'oggetto su
cui chiamiamo il metodo) e ad altroAnimale (passato come
parametro). */
public boolean equals(Animal altroAnimale) {
    return
        (this.nome.equalsIgnoreCase(altroAnimale.nome))
        &&
        (this.eta == altroAnimale.eta)
        &&
        (this.peso == altroAnimale.peso);
}
}

/* Verifichiamo che essere uguali e' diverso dall'avere lo
stesso indirizzo. Usiamo la classe AnimalDemo e il file
AnimalDemo.java */
public class AnimalDemo { //classe eseguibile pubblica
    public static void main(String[] args){
        Animal 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
    }
}

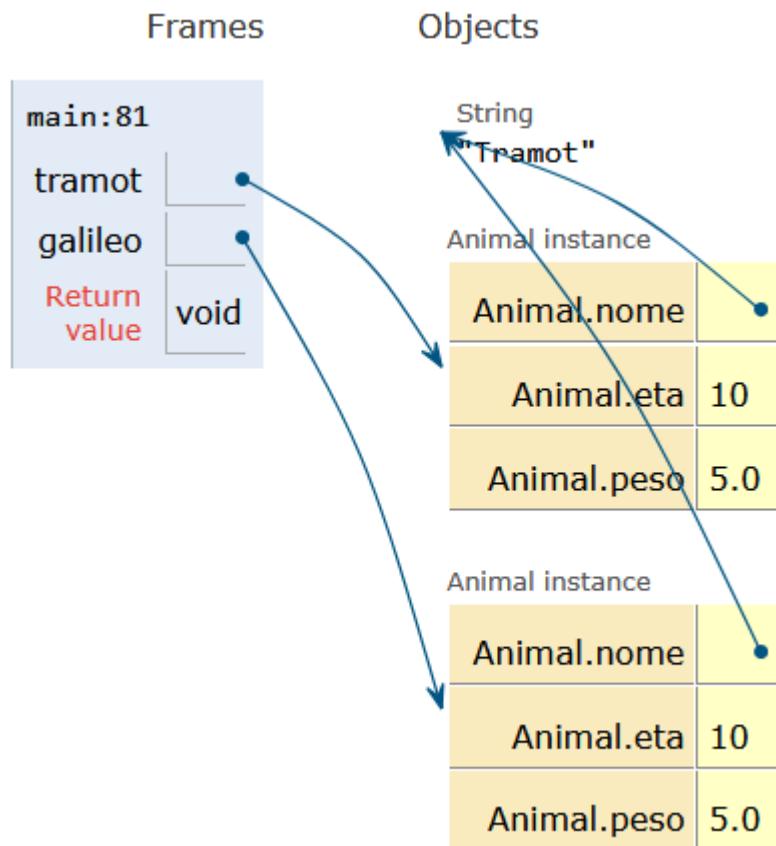
```

```

        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 i valori degli 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 modifco z non modifco 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 sostituire i valori dei campi di un oggetto passato come parametro con i valori dell'oggetto this, ovvero dell'oggetto su cui chiamiamo il metodo. Nell'esempio astratto di cui sopra, x,y,z sono, rispettivamente: **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 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
        invocato su un oggetto, ma chiamato scrivendo:
        SpecieNuovaDemo.pause();
        In realtà, poiché in questo esempio viene invocato all'interno
        della classe stessa, possiamo invocarlo direttamente così:
        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
metodo di 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
metodo di 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("Ora le due variabili puntano allo stesso
oggetto:");
System.out.println(specieTerrestre);
System.out.println(specieKlingon);

System.out.println("\n 6. Per rendermi conto che le due
variabili sono un alias dello stesso oggetto: " );
System.out.println("se modifco la specie terrestre in
Elefante modifco 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. La prima e la seconda variabile
puntano allo stesso oggetto: (specieTerrestre ==
specieKlingon) vale : "
+ (specieTerrestre == specieKlingon)
);
/** Vero, sono lo stesso indirizzo */

System.out.println("\n 9. Invece la prima e la terza
variabile no: (specieTerrestre == specieAfricana) vale : "
+ (specieTerrestre == specieAfricana)
);
/** Falso, hanno gli stessi valori ma non lo stesso indirizzo
*/

```

```

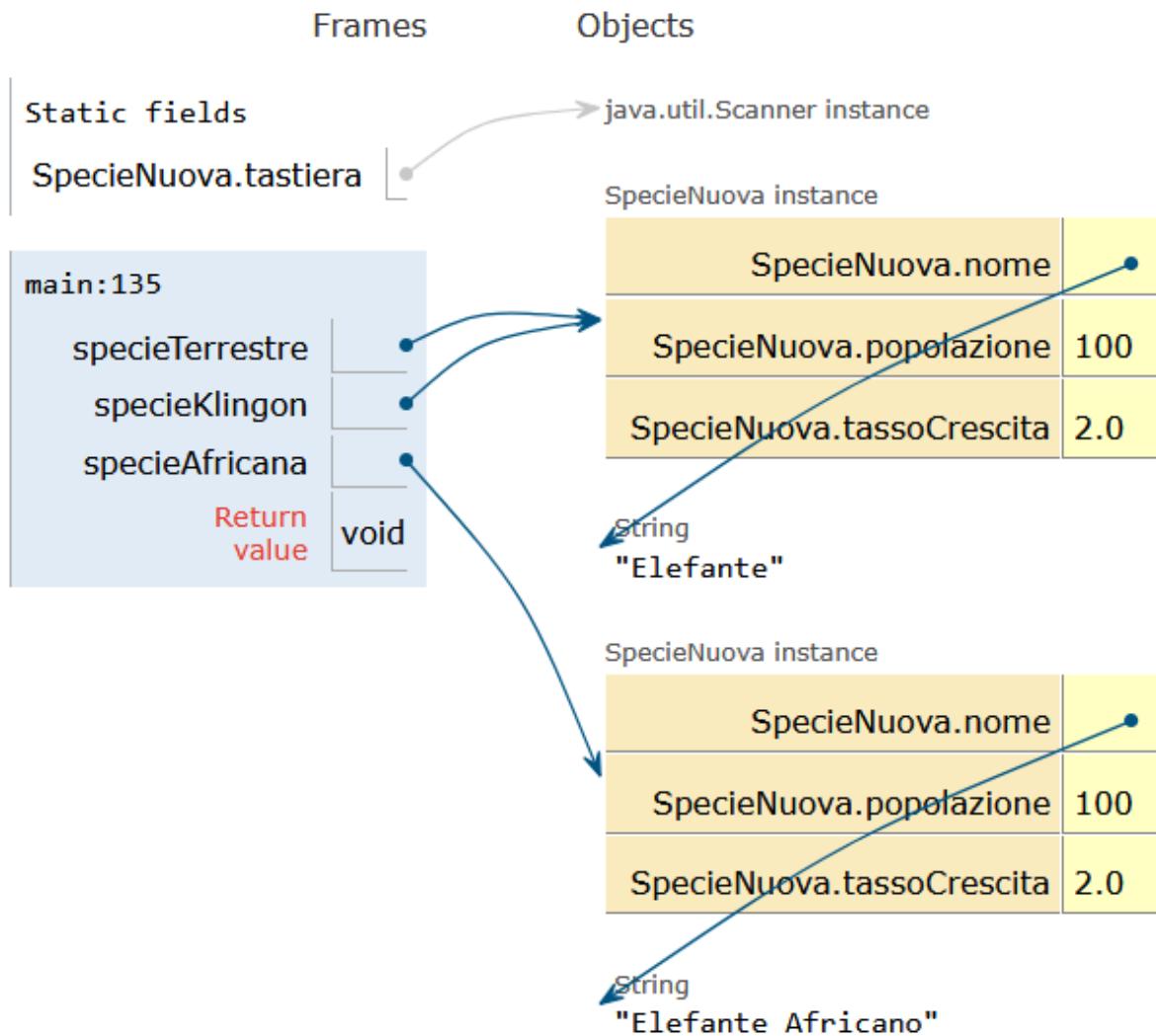
System.out.println( "\n 10. Pero' gli attributi degli
oggetti puntati dalle due variabili hanno gli stessi valori:
(specieTerrestre.equals(specieAfricana)) vale : " +
(specieTerrestre.equals(specieAfricana)));
/** Vero, hanno gli stessi valori ma indirizzo diverso */
pause();

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

System.out.println("\n 12. NON ho modificato l'oggetto
puntato da specieTerrestre e specieKlingon perche' ha un
diverso indirizzo: ");
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 quindi le modifiche sul terzo non si ripercuotono sui primi due.



**Nota sui cast in Java.** Nel metodo `prediciPopolazione()` della classe `SpecieNuova`:

```

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

```

appare un'operazione di cast esplicito: `(int)p`. In Java, questa è una direttiva al compilatore che gli impone di considerare il risultato della `return`, ovvero il valore della variabile `p` come un `int` e non come un `double` (che è invece il tipo della variabile). Questa scelta fa sì che la parte decimale venga troncata (NON arrotondata: volendo arrotondare, si deve usare il metodo statico `Math.round()` della classe

`Math13`). Si noti che c'è anche un cast *implicito* nell'istruzione:

```
double p = popolazione;
```

poiché il campo popolazione è di tipo int e invece la variabile p è di tipo double. In questo caso, non è necessario mettere un cast esplicito, perché l'insieme dei valori int è sottoinsieme dei valori double (in particolare, gli int sono tutti i double con parte decimale pari a 0), perciò un int è anche un double e quindi l'assegnamento può avvenire senza direttive al compilatore. ATTENZIONE: un double, invece, non è un int, quindi non sarebbe corretto l'assegnamento inverso, come non lo sarebbe l'istruzione return di cui sopra, senza un cast esplicito (omettendo il cast in questi casi, il compilatore dà errore).

**Soluzione dell'esercizio proposto nella Lezione 03.** La classe Rettangolo ha attributi base, altezza e area: l'area dipende da base e altezza. Se gli attributi fossero pubblici, allora un codice client li potrebbe modificare direttamente. Questo potrebbe causare inconsistenze: per esempio, si potrebbero modificare base e altezza, dimenticandosi di modificare opportunamente l'area. Se invece si hanno i campi privati e i metodi set() e get() pubblici, si fa in modo che il metodo set() provveda ad aggiornare sempre anche il campo area quando cambiano base e altezza.

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

class Rettangolo { //classe non esegibile

    /** 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;}
}
```

---

<sup>13</sup> <https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

```

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 2x2
    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
}

```

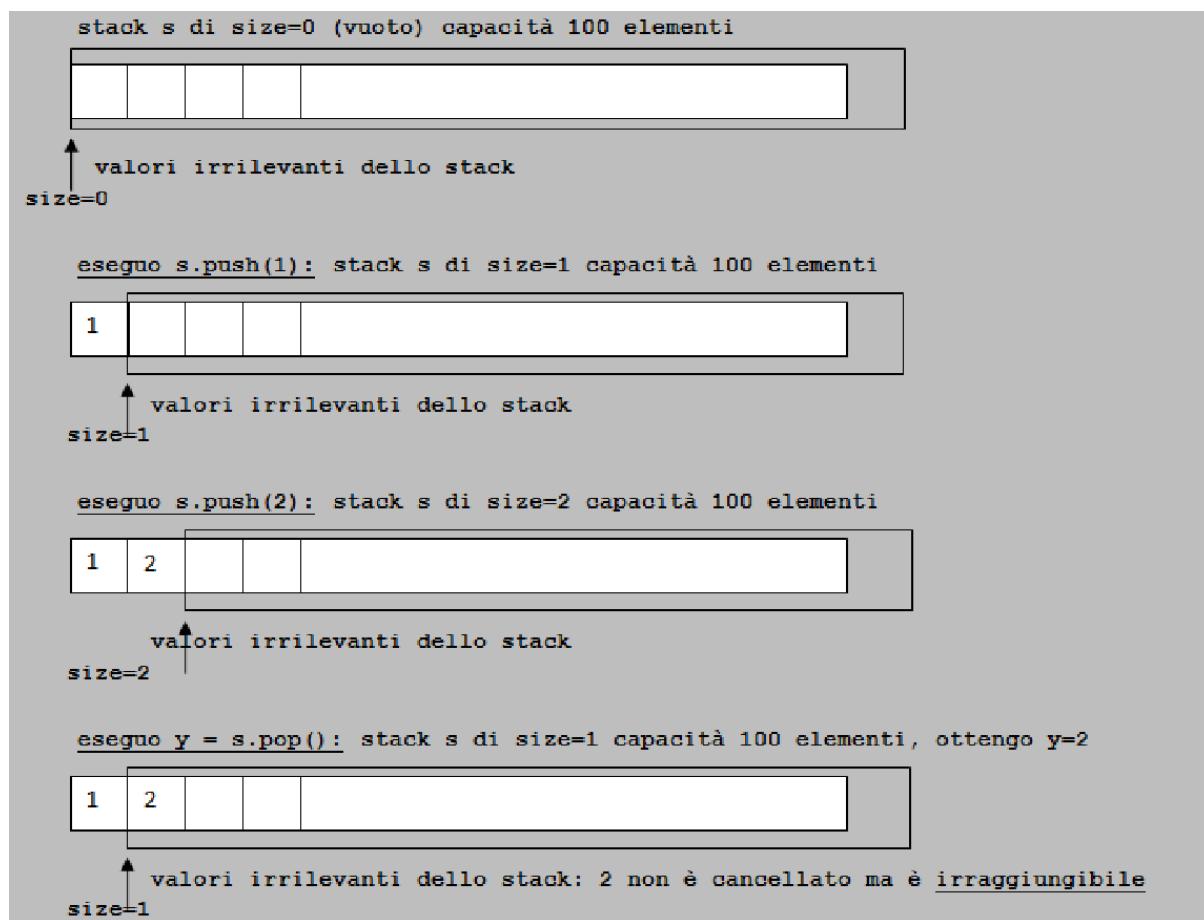
```
//r.toString()

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 05

### La classe "Stack". Chiamate di metodi

**Lezione 05. Parte 1. Un primo esempio di libreria: la classe Stack.** 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()**) 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 un'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 generale 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, così non può influire sul corretto funzionamento di uno stack, eseguendo operazioni non corrette (per esempio cambiando il valore si size al di fuori delle operazioni push e pop). 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 (l'IDE con cui forse 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 Stack.java

public 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
il programma non fallisca a runtime se vengono inseriti dati
sbagliati. A tal fine impareremo a usare il meccanismo delle
'eccezioni'. Per ora, ci concentriamo a trovare i casi in cui
un programma potrebbe fallire a runtime e usiamo le assert
come primo meccanismo per segnalare questi casi di errore a
tempo di esecuzione. */

// Adesso fissiamo: massimo numero elementi stack = capacity
stack = new int[capacity];
// size = numero di elementi inseriti; all'inizio e' 0
size = 0;
}

// È 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
quegli
di indice 0, ..., size-1. */
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 è 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));

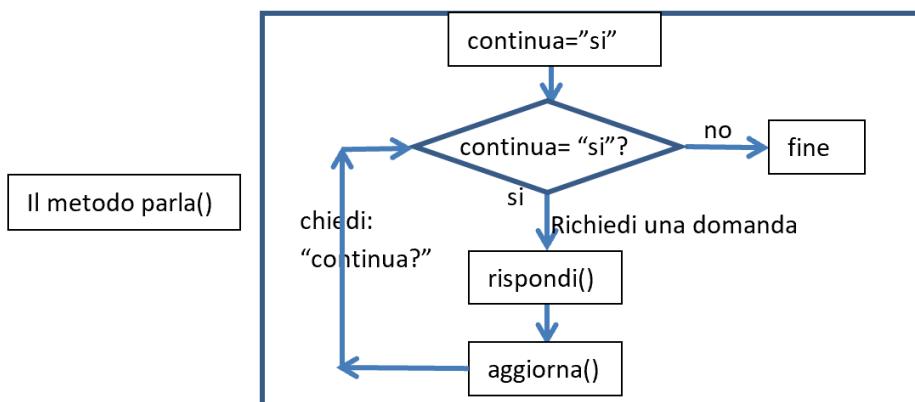
    Stack r = new Stack(3);
    r.push(0); r.push(0); r.push(0);
    Stack q = new Stack(3);
    System.out.println("r non è vuoto!");
    System.out.println(" r.full()      = " + r.full());
    System.out.println(" r.empty()     = " + r.empty());
    System.out.println(" r.equals(q) = " + r.equals(q));
    /* 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, perché la sua size è 0).
    */
}
}

```

**Lezione 05. Parte 2. Chiamate tra metodi.** I metodi dinamici si possono richiamare gli uni gli altri. L'unico vincolo è che

queste catene di chiamate devono essere in numero finito perché il programma termini. 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" e una "domanda" (oltre a un oggetto di tipo Scanner per leggere gli input). **Si noti come alcuni metodi della classe Maestro richiamano altri metodi della stessa classe.** 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 al
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 è 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?

si

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

## Lezione 06

### Modelli di oggetti reali. Information Hiding

**Lezione 06. Parte 1. Metodi statici e dinamici.** Alcune funzionalità si implementano più facilmente con **variabili** (dette anche attributi o campi) e **metodi statici**, ovvero non legati a oggetti.

Le variabili statiche sono "globali" a una classe, ovvero corrispondenti ciascuna a un'unica locazione di memoria (quindi non una copia per ogni oggetto che viene istanziato, come invece succede per le variabili dinamiche). Le variabili statiche non vengono allocate né nello stack, né nella heap, ma in un'area di memoria detta **area delle classi**.

Per introdurre una variabile/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.

**Proprietà invariante per la classe Bottiglia.** 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 più di un'invariante. Per la classe Bottiglia, un'invariante è che il livello del liquido nella bottiglia deve essere sempre  $\geq 0$  e (contemporaneamente) minore o uguale alla capacità della bottiglia: **0  $\leq$  capacita AND 0  $\leq$  livello  $\leq$  capacita**.

```
//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){
```

```

    assert (0 < 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" e' un attributo di bottiglia mentre
    "livello" e' l'unico argomento del metodo setLivello() */
    // assert sul parametro:
    assert (0 <= livello)&&(livello <= capacita): "il valore del
    livello da impostare nella bottiglia deve essere >=0 e
    contemporaneamente non superiore alla capacita` della
    bottiglia";
    this.livello = livello;
    //assert sul campo:
    assert (0 <= this.livello)&&(this.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 operazioni (implementate 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 (non avrebbe senso).
    Dunque, i metodi di DieHard sono dichiarati statici e non
    vengono chiamati su oggetti della classe (si veda nel
    main()).*/
}

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

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

public static void travasa(Bottiglia a, Bottiglia b){
    // calcolo quanto basta per riempire b
    // 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 06. Parte 2. Un altro esempio di encapsulamento di dati: la classe **Frazione**.** Riprendiamo l'argomento di encapsulamento dei dati, e definiamo una classe **Frazione** per rappresentare le frazioni intere e le operazioni su di esse. Per semplicità di rappresentazione, useremo solo frazioni **con denominatore positivo**.

I calcoli su frazioni tendono a produrre numeratori e denominatori molto grandi: per evitarlo, nella nostra implementazione una frazione viene sempre **ridotta ai minimi termini**. Per evitare che chi usa la classe si dimentichi di questa regola, rendiamo privati gli attributi numeratore e denominatore. Inoltre, inseriamo un metodo `set()` che assegna **contemporaneamente** numeratore e denominatore e semplifica la frazione. I metodi `get()` e `set()` per il numeratore e il denominatore (presi individualmente) non sono necessari in quanto manipoliamo la frazione come un tutt'uno attraverso i metodi che implementano le operazioni tra frazioni.

Una frazione viene definita come una espressione `(num/den)` con `den>0`:

- Per ridurre una frazione ai minimi termini dividiamo numeratore e denominatore per il Massimo Comune Divisore `MCD(num,den)`: per esempio,  $6/4$  diventa  $3/2$ .
- Cambiamo segno a entrambi se il numeratore è negativo:  $1/(-2)$  diventa  $(-1)/2$ .
- Calcoliamo la somma di due frazioni con la formula 
$$\frac{a}{b} + \frac{c}{d} = \frac{(ad + bc)}{bd}$$
- Calcoliamo il prodotto di due frazioni con la formula 
$$(\frac{a}{b}) (\frac{c}{d}) = \frac{(ac)}{(bd)}$$
.

- 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.** Come già specificato, 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 più di un'invariante. 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 Comune 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 (versione con il this esplicito) */
    private void semplifica(){
        int m = MCD(this.num, this.den);
```

```

this.num = this.num/m;
this.den = this.den/m;
if (this.den<0){this.num = -this.num; this.den = -this.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. */
```

```
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 07

### Classi di array di oggetti

In questa lezione implementeremo una classe Contatto, che astrae il concetto di "contatto" tramite una coppia di attributi che rappresentano un nome e un indirizzo e-mail, e una classe Rubrica, che rappresenta una lista di contatti. La classe Rubrica è un esempio di classe in cui la struttura interna dei suoi oggetti (oggetti di tipo Rubrica) è un **array di oggetti**. A loro volta, gli elementi dell'array sono oggetti di tipo Contatto.

La classe Contatto ha attributi privati, metodi get() e set() senza restrizioni e un metodo toString().

```
// Contatto.java

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 oggetto Contatto nella stringa che lo
    descrive
    public String toString()
    {return " - " + nome + " : " + email;}
}

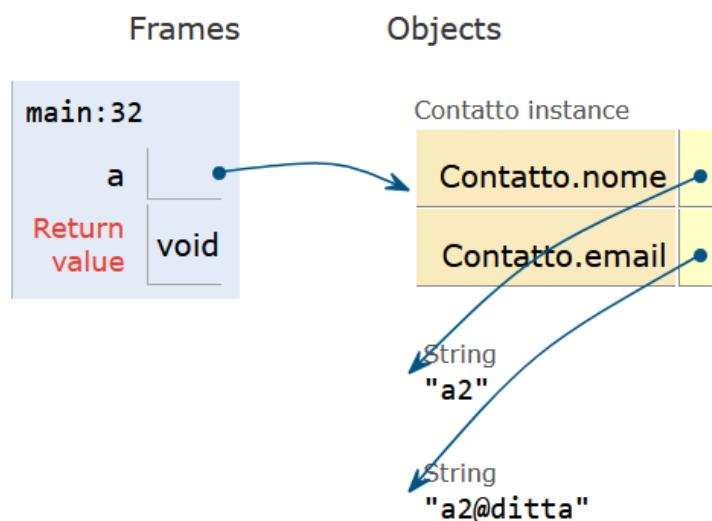
// ContattoDemo.java
public class ContattoDemo {
    // controllo i metodi della classe Contatto
    public static void main(String[] args){
        Contatto a = new Contatto( "a", "a@ditta");
    }
}
```

```

System.out.println( "Contatto a: " + a );
System.out.println( "Modifico nome a in a2" );
a.setNome( "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 disegno si riferisce a questo punto della
computazione
String nomea= a.getNome();
String emaila= a.getEmail();
// ESERCIZIO: stampare con println() nomea e emaila
}
}

```

Segue uno snapshot dell'esecuzione del main() di ContattoDemo.



L'oggetto **Contatto** con **nome** "a" e **email** "a@ditta", dopo che abbiamo modificato nome ed e-mail. La variabile a si trova sullo stack e contiene l'indirizzo dell'area della heap che contiene nell'ordine nome e email.

La classe **Rubrica** ha come attributi privati:

1. un array di oggetti di tipo Contatto (campo **contatti**), con posizioni da 0 a maxContatti-1 (dove **maxContatti** rappresenta la lunghezza massima dell'array ed è parametro del costruttore), e
2. un campo intero **numContatti**, che dice quanti contatti sono stati effettivamente inseriti nell'array. L'array è

quindi **parzialmente riempito**, poiché il numero degli elementi effettivi è:  $0 \leq \text{numContatti} < \text{maxContatti}$ .

Ricordiamo che l'**invariante** di una classe è una proprietà inizialmente vera per ogni oggetto costruito e che deve essere preservata da ogni applicazione di ogni metodo. Definiamo Rubrica in modo da rendere vera la seguente invariante: **i posti occupati vanno da 0 a numContatti-1, ogni nome compare in al più un contatto di una rubrica, e ( $0 \leq \text{numContatti} \leq \text{lunghezza array contatti}$ )**. 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 un errore di programmazione non rispettassimo questa parte dell'invariante** a un nome corrisponderebbero più email, e gli oggetti di tipo Rubrica **non funzionerebbero secondo le specifiche date**: il mantenimento dell'invariante di classe nel codice è **essenziale per il corretto funzionamento degli oggetti**.

**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 l'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(). Non mettiamo il metodo setContatti(), che modifica l'array, per un motivo analogo. Non mettiamo il metodo getContatti() (che restituisce l'indirizzo dell'array, poiché l'array è un oggetto e come tale sta nella heap) per evitare *memory leak*, di cui parleremo più avanti.

**Rubrica ha un metodo privato int cercaIndice(String n)** per ottenere l'unica posizione di un nome nell'array contatti, che è un intero tra 0 e numContatti-1. Restituisce numContatti se il nome non compare. Per l'identificazione dei contatti si usano i nomi ignorando le differenze dovute alla presenza di lettere maiuscole/minuscole. Il metodo cercaIndice() viene usato per implementare i metodi pubblici della classe. Abbiamo scelto di implementarlo come privato perché non vogliamo che il codice client abbia accesso diretto agli elementi

dell'array che rappresenta la struttura della rubrica: se ce l'avesse, potrebbe cambiare l'oggetto nella posizione restituita da `cercaIndice()` senza rispettare l'invariante.

Rubrica ha metodi pubblici:

- (i) **`String toString()`** per costruire una stringa che descrive la rubrica,
- (ii) **`boolean presente(String n)`** per decidere se un nome è presente nella rubrica,
- (iii) **`String cercaEmail(String n)`** per cercare un indirizzo e-mail dato il nome (restituisce "", ovvero stringa vuota, 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 preservata tramite un'assert. Se è così, l'azione viene eseguita e restituiamo **true**, se non è così l'azione non viene eseguita e restituiamo **false**.

Ecco il codice completo della classe Rubrica.

```
//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. numero di contatti =
        maxContatti
        contatti = new Contatto[maxContatti];
        numContatti = 0;
```

```

//all'inizio i contatti effettivi nella rubrica sono 0

//IMPORTANTE: all'inizio tutti gli elementi dell'array
//contatti (ovvero i contatti nella rubrica) hanno valore
null
/** Dopo l'esecuzione del costruttore, l'oggetto di tipo
Rubrica soddisfa l'invariante. */
}

public int getNumContatti(){return numContatti;}
/** non definiamo un metodo getContatti() 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 tale contatto e' nella rubrica.
Altrimenti restituisce numContatti. Il metodo cercaIndice(n)
e' privato in quanto unicamente di servizio per gli altri
metodi della classe. Non serve all'esterno della classe in
quanto non abbiamo definito metodi per accedere agli elementi
dell'array, peraltro non accessibile (perche' private e senza
metodo accessorio getContatti()). */
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 di default:
numContatti
    return numContatti;
}

/** Usando cercaIndice(n) possiamo stabilire se il nome n è
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 "";
}

/** Controlloiamo se una rubrica è piena, cioè se il
numContatti è 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 numContatti e scambiamo il contatto da cancellare
che sta in posizione i con il contatto che sta in posizione
numContatti. In questo modo, il contatto che era prima in
posizione i non sarà più accessibile, visto che la posizione
numContatti rappresenta la prima posizione libera nell'array.
Si noti che questa cancellazione non è fisica (non si rimuove
l'oggetto dalla struttura), ma logica: si fa in modo che
l'oggetto da cancellare non sia più raggiungibile utilizzando
i metodi di gestione della rubrica.

Inoltre, questa cancellazione non preserva l'ordine dei
contatti. Tuttavia nell'invariante non c'è scritto che la
rubrica deve rispettare un certo ordinamento quindi non ci
sono problemi. */
    --numContatti;
    contatti[i] = contatti[numContatti];
    return true; //successo
}

// Cerco un contatto di nome n e se lo trovo (e nessun
contatto in rubrica ha nome n2) 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.

```

Si noti che, per come abbiamo definito Rubrica, ammettiamo che ci siano più contatti con la stessa email (non abbiamo inserito nell'invariante la condizione di unicità delle mail). Pertanto, il metodo cambiaEmail() non fa verifiche sul valore del parametro e2. \*/

```

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

```

Ora presentiamo il codice di una classe di test, RubricaDemo.java.

```

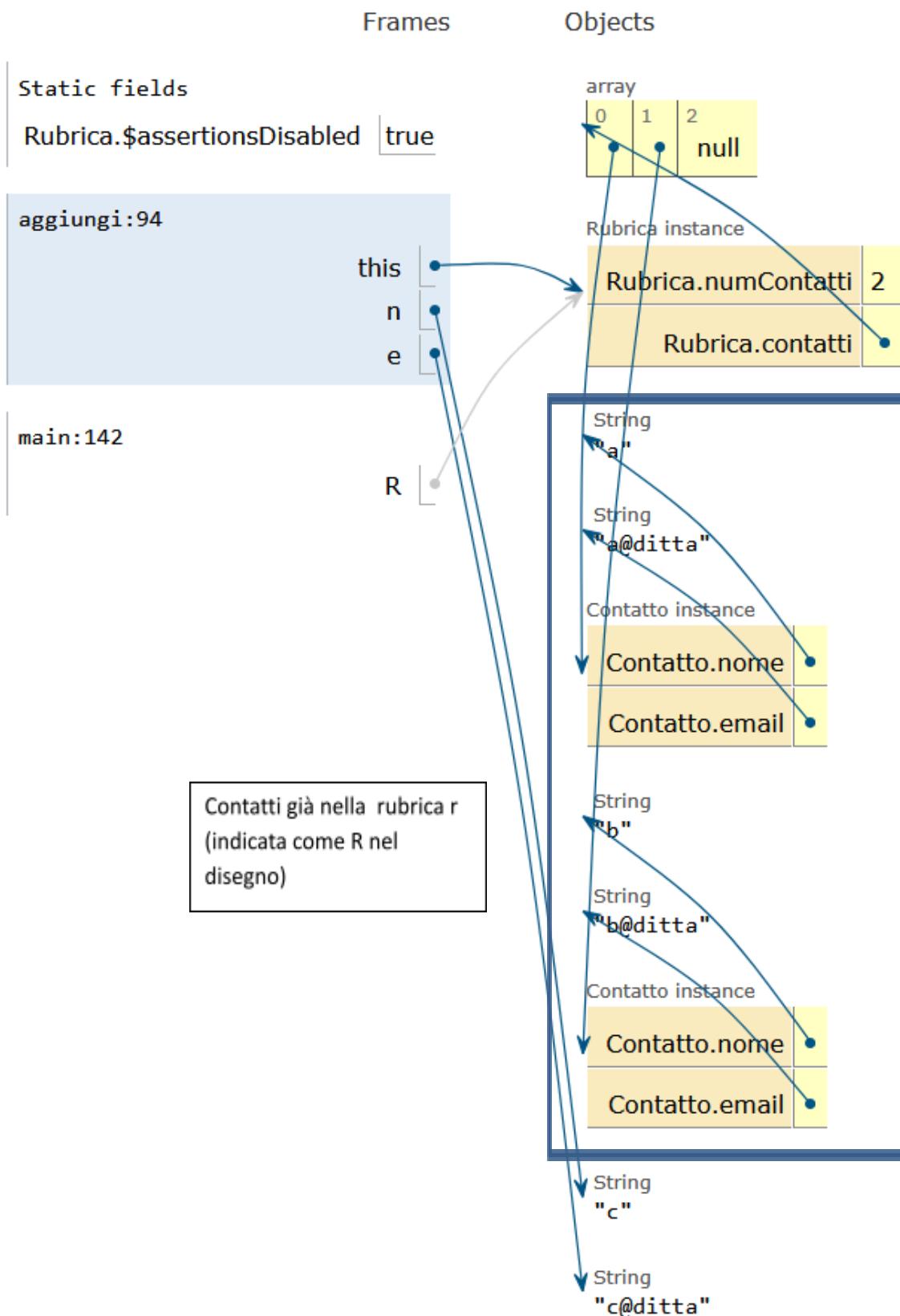
//RubricaDemo.java
public class RubricaDemo {
    public static void main(String[] args){
        //Consentiamo 3 elementi in rubrica e proviamo a inserirne 4
        Rubrica r = new Rubrica(3);
        System.out.println("(1) Rubrica con contatti a,b,c: " );
        r.aggiungi( "a", "a@ditta"); r.aggiungi( "b", "b@ditta");
        r.aggiungi( "c", "c@ditta"); r.aggiungi( "d", "d@ditta");
        System.out.println(r);
        //vengono stampati 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.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);
    }
}

```

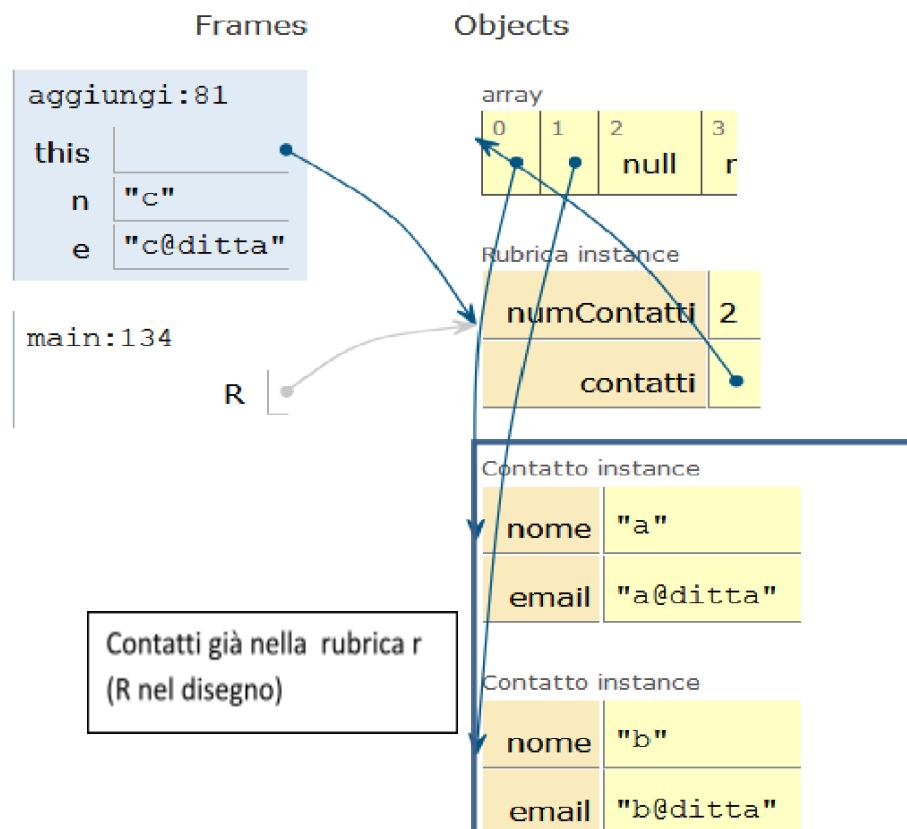
Seguono due snapshot costruiti con il Java Visualizer. Tutti e due rappresentano la memoria della JVM durante la stessa

esecuzione del metodo aggiungi(). La differenza è che nel primo le stringhe sono rappresentate correttamente come oggetti memorizzati nella heap; invece nel secondo abbiamo scelto di scriverle direttamente nelle variabili sullo stack e nei campi. Questa seconda rappresentazione contiene dunque un'inesattezza, che però è stata introdotta per rendere la figura più leggibile. **D'ora in poi, considereremo questa inesattezza come una convenzione e la applicheremo in tutte le figure.**

## Primo snapshot



## Secondo snapshot



**L'effetto del metodo `aggiungi()` sulla memoria.** Il metodo `aggiungi()` aggiunge l'oggetto di tipo `Contatto` di nome "c" e email "c@ditta" all'oggetto di tipo `Rubrica` puntato dalla variabile `r` che contiene già gli oggetti `Contatto` con nomi "a" e "b". Nella parte dello stack relativa al metodo `aggiungi()`, `this` che diventa uguale a `r` punta a un'area di memoria che include la dimensione dell'array, e l'indirizzo dell'array. L'array contiene gli indirizzi attuali degli oggetti `Contatto` di nome "a" e "b". A loro volta, gli oggetti di tipo `Contatto` sono coppie di informazioni nome/email di tipo `String`.

### Nota avanzata su: rimozione di un contatto da una rubrica e alias

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. Se assegniamo `y = x`, dopo l'assegnamento `x` e `y` contendono l'indirizzo della stessa area di memoria, quindi indicano lo stesso oggetto.

In altre parole, x e y possono essere usate per modificare lo stesso oggetto in memoria. Due indirizzi x e y per lo stesso oggetto nell'area raggiungibile della memoria vengono detti **alias**.

Possiamo chiederci se il metodo **boolean rimuovi(String n)** (visto in questa lezione), che rimuove un contatto da una rubrica, crei indirizzi duplicati per lo stesso oggetto di tipo Contatto, ovvero degli "alias". 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 solo uno di questi è raggiungibile da un punto di vista logico*, cioè dal punto di vista del corretto funzionamento della rubrica. Infatti, il metodo **rimuovi()** trova un indice i in cui compare l'oggetto Contatto c\_i di nome n e lo cancella **logicamente**

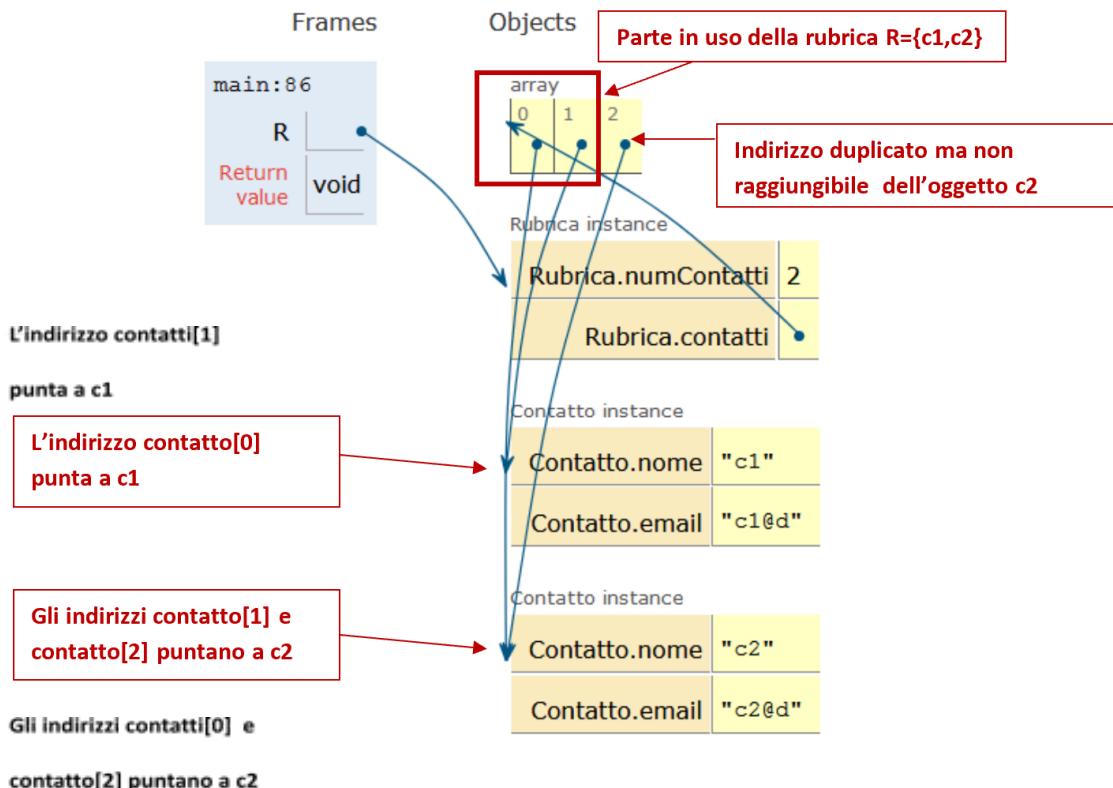
- riducendo di uno il numero dei contatti numContatti
- e scambiando il contatto c\_i con l'ultimo contatto effettivo (quello in posizione numContatti dopo il decremento).

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]. A questo punto, c0 è sovrascritto, contatto[0] contiene l'indirizzo di c2, contatto[1] contiene l'indirizzo di c1 e contatto[2] contiene l'indirizzo di c2.

Abbiamo quindi creato due alias, dato che contatto[0] e contatto[2] contendono entrambi l'indirizzo di c2. **Tuttavia l'alias contatto[2] non è logicamente raggiungibile** dai metodi basati sulla ricerca di un oggetto nell'array **contatti[]**: la variabile contatto[2] è irraggiungibile dal punto di vista logico perché si trova nella posizione 2. Dato che numContatti=2, la posizione 2 non fa parte delle posizioni che

risultano occupate. Rappresenta invece la prima posizione libera nell'array.

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



È possibile che in futuro si possa raggiungere l'indirizzo duplicato dell'oggetto c2, quello che si trova in contatto[2]? No, non si può. Potrei aggiungere un nuovo oggetto c'2 alla rubrica, riportando numContatti a 3. Ma, in questo caso, contatto[2] diventerebbe c'2, l'indirizzo di un nuovo oggetto, e l'indirizzo c2 in contatto[2] verrebbe sovrascritto. In nessun caso il secondo indirizzo c2 contenuto in contatto[2] può venir utilizzato da un programma che usa la classe Rubrica, quindi questo secondo indirizzo non costituisce un alias che possiamo effettivamente usare.

# Lezione 08

## Diagrammi UML e array estendibili

**Lezione 08. Parte 1. La descrizione delle classi con diagrammi.** Scrivere programmi è un lavoro di gruppo e non di singoli, e lo stesso programma può essere ripreso e modificato da persone diverse. Questo comporta la necessità di fissare un linguaggio comune per descrivere e documentare le caratteristiche generali del software durante la progettazione e la programmazione. Per il paradigma a oggetti il formalismo più comunemente usato è grafico e si chiama **UML** (**Unified Modeling Language**). In questa lezione introdurremo uno dei diagrammi UML più usati, il *class diagram*, che useremo per descrivere le classi dell'esempio-guida.

**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 gli attributi e i metodi statici sono sottolineati<sup>14</sup>. Come esempio prendiamo la classe Matita vista in laboratorio.

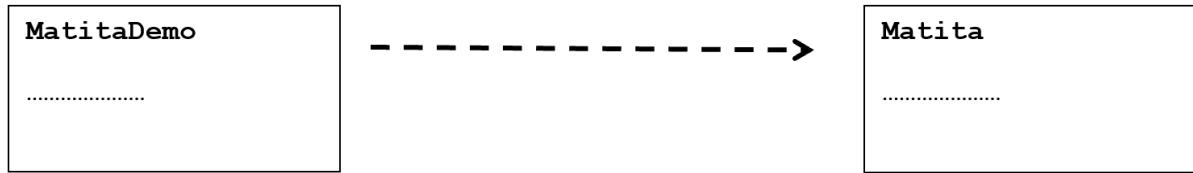
| Matita                   |
|--------------------------|
| + <u>minStelo</u> :int   |
| + <u>maxStelo</u> :int   |
| + <u>maxPunta</u> :int   |
| - stelo :int             |
| - punta :int             |
| + Matita(stelo:int)      |
| + disegna() :void        |
| + tempera() :void        |
| + <u>getStelo()</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 quindi che le modifiche di D possono ripercuotersi su C. Questa relazione viene detta di **dipendenza**. Per esempio la

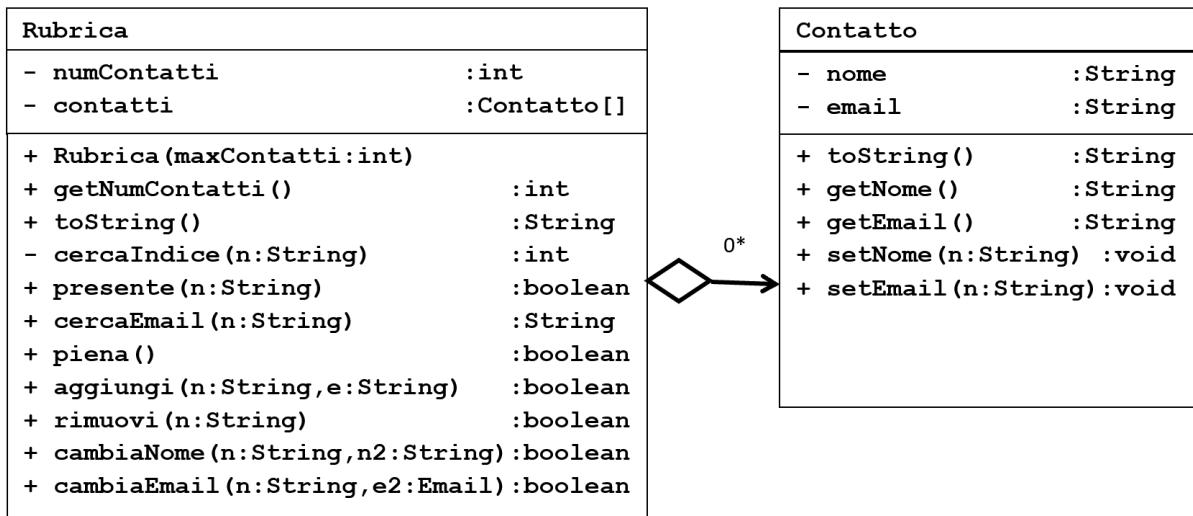
---

<sup>14</sup> A volte le componenti statiche sono indicate in corsivo.

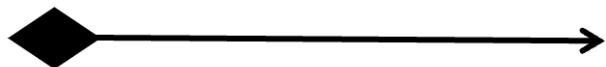
definizione di un client MatitaDemo dipende dalla definizione di Matita:



Una relazione più specifica 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 07) **aggrega la definizione di Contatto** (quindi, si ha C = Rubrica e D = Contatto). Con l'annotazione 0\* indichiamo che una rubrica può contenere un numero qualsiasi di contatti (anche zero se è vuota).



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 08. Parte 2. Array estendibili.** Definiamo una classe di array parzialmente riempiti, con la possibilità di aggiungere e togliere elementi. Per risolvere il problema della dimensione fissa degli array, implementiamo una classe che permette di gestire “array estendibili” i quali cambiano dimensioni per ospitare un numero qualunque di elementi. Poiché, tuttavia, gli array di Java hanno dimensione fissa, la capacità di estendersi può solo essere realizzata sostituendo l’array originale con uno di lunghezza maggiore. Pertanto, nella nostra implementazione, 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 dal garbage collector.

**La classe ArrayExt.** Come già visto per altre strutture dati nelle precedenti lezioni, oltre a un campo array, nella classe definiamo un campo size che rappresenta il numero di elementi effettivi memorizzati nell’array (posizioni 0, ..., size-1).

L’**invariante** di classe è **(0 ≤ size ≤ lunghezza\_array)**. Il costruttore impone la lunghezza iniziale dell’array e assegna il campo size a 0. Abbiamo metodi pubblici per:

- (i)** Ottenere size dell’array estendibile.
- (ii)** Restituire il contenuto dell’array in formato stringa.
- (iii)** Ottenere l’elemento di posto  $0 \leq i < \text{size}$ .
- (iv)** Assegnare l’elemento di posto  $0 \leq i < \text{size}$ .
- (v)** Inserire un elemento in posizione  $0 \leq i \leq \text{size}$  (size incluso), spostando avanti di una posizione tutti i successivi elementi, e aggiungendo 1 a size.
- (vi)** Rimuovere un elemento in posizione  $0 \leq i < \text{size}$  (size escluso), spostando indietro di una posizione tutti i successivi elementi, e togliendo 1 a size. L’elemento rimosso viene restituito come risultato.

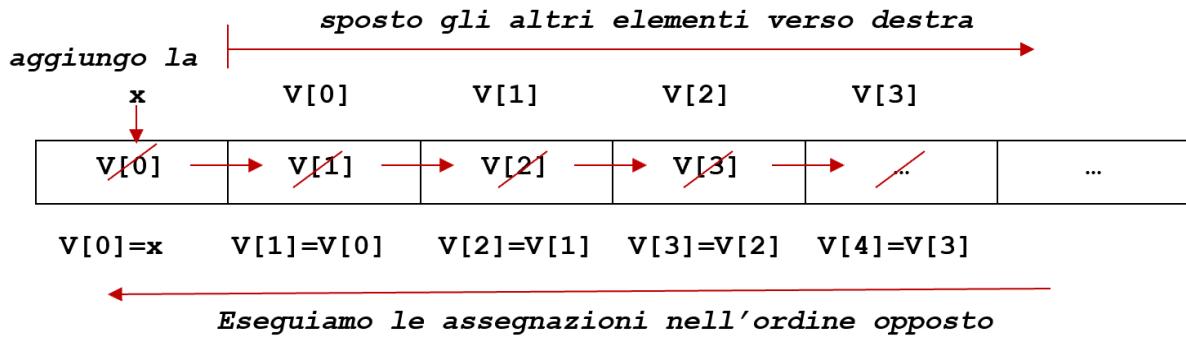
**(vii)** Raddoppiare la dimensione dell'array: questo è un metodo privato, utilizzato dalla classe quando aggiungere un elemento farebbe superare a size la lunghezza dell'array.

Si noti che scegiamo di implementare l'inserzione e la rimozione degli elementi tramite uno spostamento rispettivamente in avanti e indietro degli elementi stessi in modo che, se esiste un ordinamento tra essi, l'ordinamento venga mantenuto.

#### Diagramma UML di ArrayExt

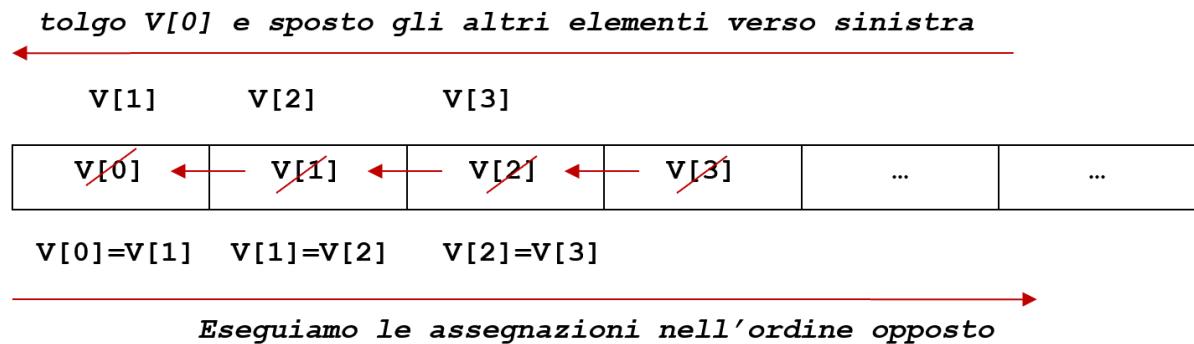
| <b>ArrayExt</b>            |         |
|----------------------------|---------|
| - <b>size</b>              | :int    |
| - <b>vett</b>              | :int[]  |
| + <b>ArrayExt(min:int)</b> |         |
| + <b>getSize()</b>         | :int    |
| + <b>toString()</b>        | :String |
| + <b>get(i:int)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

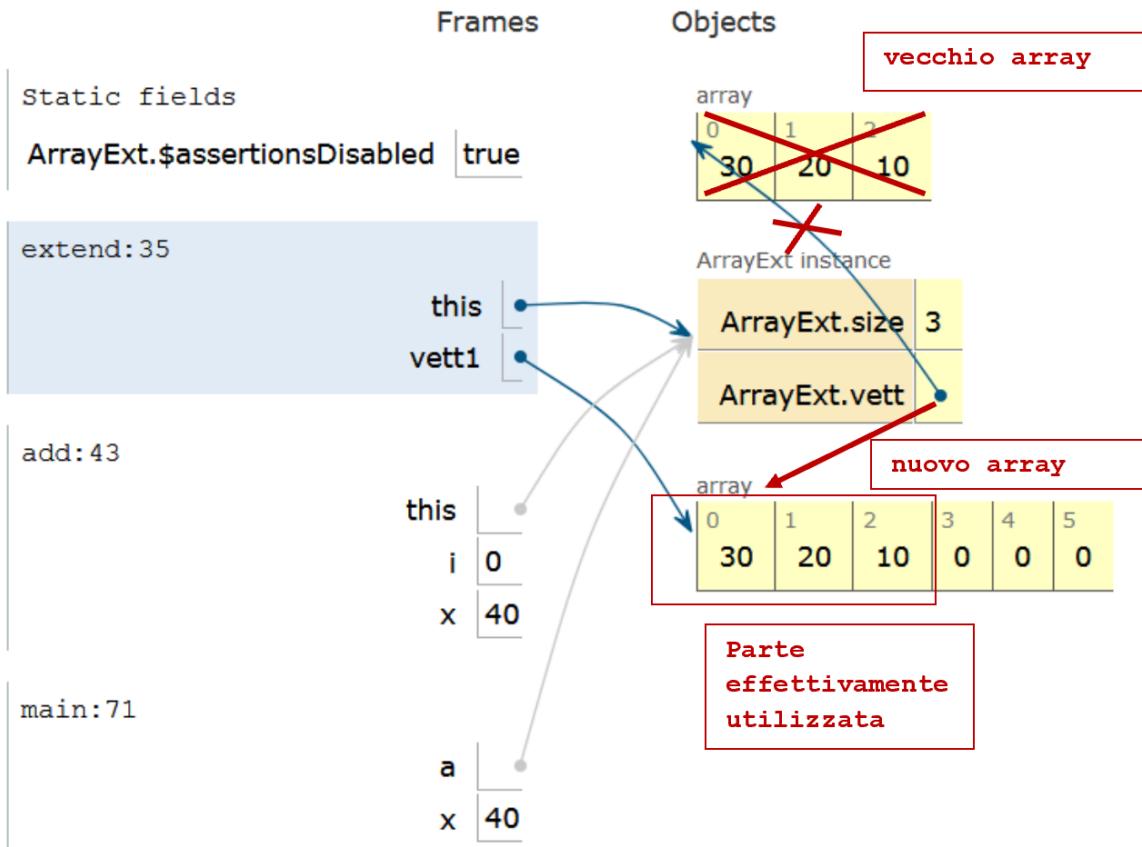


**Nota.** Se invece assegnassimo  $v[1]=x$ ,  $v[2]=v[1]$ ,  $v[3]=v[2]$ ,  $v[4]=v[3]$ , muovendoci da sinistra a destra, otterremo:  $x$ ,  $x$ ,  $x$ ,  $x$ . **Non faremmo altro che ricopiare  $x$  più volte: non fate così!**

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:



**Vediamo un esempio concreto.** Aggiungere un elemento 40 nella posizione 0 di un array `vett={30,20,10}` pieno (chiamando un metodo `add()`) ha come primo effetto la chiamata al metodo di duplicazione dell'array (metodo `extend()`), che ha il compito 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
/* La 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 mantenendone l'ordinamento. \*/

```
public class ArrayExt{
```

```
private int size;
private int[] vett;
```

```

// Invariante: (0 <= size <= lunghezza vett) e (lunghezza
vett>0) (ovvero parte significativa di vett: da 0 a size-1)

// METODI

public int getSize(){return size;}

/* Se min>0, il seguente costruttore costruisce un array di
min elementi con size=0. Da qui in avanti, la lunghezza di
vett sara': min*(2^n), n >= 0. */
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[this.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 del campo vett
    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. Fa da
//push se l'inserimento viene fatto nel posto 0 dell'array.
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
        this.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;
}

//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;
    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.** Le operazioni appena viste sono ``classiche''. Noi le abbiamo implementate da zero a scopo didattico, ma di solito si trovano già fatte nelle librerie dei linguaggi di programmazione. Per esempio, il metodo per copiare un array in un nuovo array, possibilmente più lungo, esiste nella libreria **Arrays**, è un metodo statico e si chiama **copyOf()**. La versione che corrisponde al nostro metodo extend() ha firma

```
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() utilizzando questo metodo 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 09

### Security Leak.

#### Le classi Node e DynamicStack

**Lezione 09. Parte 1.** Un *data leak* (*data leakage* o *divulgazione di dati*) espone involontariamente informazioni sensibili, protette o riservate al di fuori dell'ambiente a cui sono destinate. Ciò accade per vari motivi, come errori umani interni, vulnerabilità del software o scarse misure di sicurezza dei dati.<sup>15</sup>

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 permette a classi esterne di modificare il dato attraverso tali *alias*.

Come esempio, definiamo una classe **CoppieAnimali** consentendo solo l'accesso **get()** (in lettura) ai due oggetti di tipo **Animale** che costituiscono i campi *primo* e *secondo* della coppia, senza accesso **set()** (in scrittura) a tali campi.

Segue il codice delle due classi.

```
//Animale.java
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)
```

---

<sup>15</sup> Tratto da <https://www.proofpoint.com/it/threat-reference/data-leak>, visitato il 17/03/2025.

```

        {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. */

    public CoppiaAnimali(String n1, int e1, double p1, String
n2, int e2, double p2){
        /** Si devono fare due "new" per ottenere oggetti diversi da
        null nei campi primo e secondo, altrimenti scrivendo, per
        esempio, primo.setNome(n1); produrrebbe a run time una
        NullPointerException.*/
        primo = new Animale(n1,e1,p1);
        secondo = new Animale(n2,e2,p2);
    }

    public Animale getPrimo()    {return primo;}
    /* Restituisce l'indirizzo del primo oggetto Animale.
    Pertanto, mette la classe che invoca il metodo in condizioni
    di modificare i dati di primo direttamente, accedendo ai campi
    dell'oggetto. */

    public Animale getSecondo() {return secondo;}
    /* idem come per getPrimo(). */

    /** VERSIONE ALTERNATIVA DELLE get(), che previene il
    problema del Security Leak.

    Se non vogliamo consentire di modificare gli attributi
    dei due animali dall'esterno, i due metodi get() devono creare
    una copia o "clone" dei due oggetti nei campi primo e secondo,
    in modo da non restituire l'indirizzo originario, ma quello
    dei cloni. Restituire un clone significa restituire un oggetto

```

```

che ha nei propri campi valori uguali a quelli dell'oggetto
originale, ma è un'istanza differente di Animale.*/

// get() implementate come 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();}

}

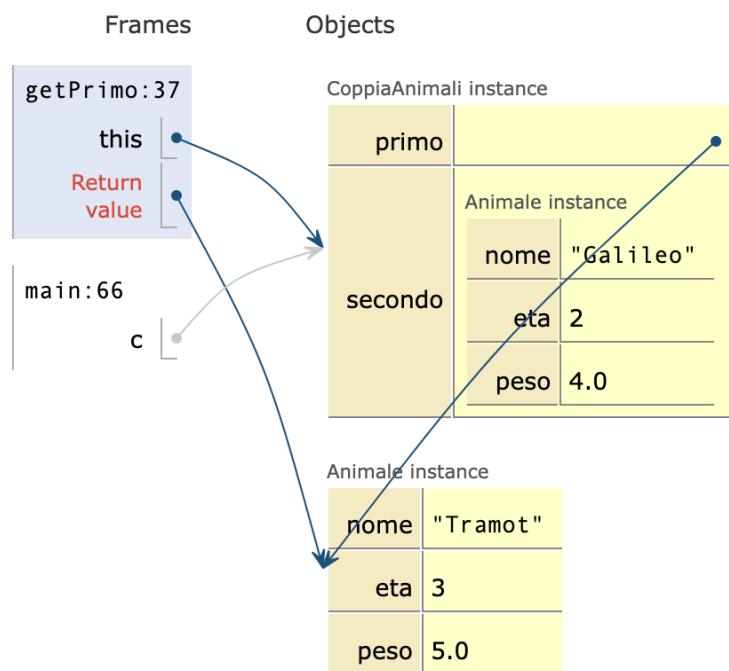
//Hacker.java (Vediamo come aggirare i divieti di scrittura)
public class Hacker {
   public static void main(String[] args){
      System.out.println("\n Definisco la coppia c con nomi
Tramot e Galileo");
      CoppiaAnimali c =
         new CoppiaAnimali("Tramot",3,5.0,"Galileo",2,4.0);
      System.out.println(c);
      System.out.println("\n Facciamo una getPrimo().");
      Animale t = c.getPrimo();
      System.out.println("Siccome getPrimo()-prima versione
restituisce una copia dell'indirizzo, possiamo modificare il
campo primo di c tramite la variabile t (suo alias).");
      System.out.println("(Se invece usiamo la
getPrimo()-versione che clona l'oggetto di c, si modifica il
clone e non l'oggetto Animale contenuto in c. In altre parole,
non si puo' modificare il campo primo di c tramite la
variabile t).");
      t.setNome("X");
      System.out.println( "\n Ora la coppia c vale \n" + c);
      System.out.println( "Ovviamente l'oggetto puntato da t
viene modificato in ogni caso." );
      System.out.println(t);
   }
}

```

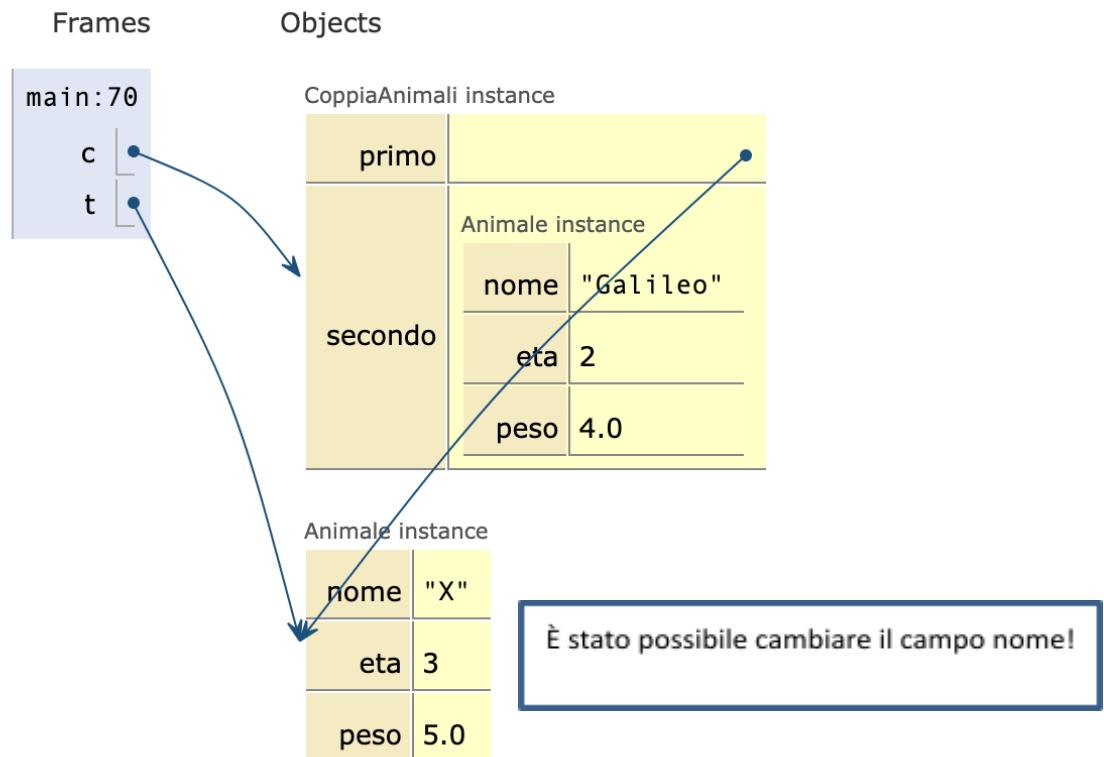
Riassumiamo ora quanto succede nel codice client contenuto nella classe Hacker. Il metodo `getPrimo()` di `CoppiaAnimali` che clona l'oggetto (e analogamente per `getSecondo()`) non consente di modificare direttamente l'indirizzo dell'oggetto contenuto nel campo "primo" originario. Tuttavia, siccome restituisce il contenuto del campo, fornisce una copia dell'indirizzo, che salviamo nella variabile `t`. Basta passare questo alias `t` a uno dei metodi `set()` (di scrittura) della classe `Animale` per modificare i campi dell'oggetto puntato dal campo `primo`.

I prossimi due disegni riassumono come è avvenuto il Security Leak: un diverso percorso attraverso i dati ha consentito di aggirare i divieti di scrittura.

**Alla fine dell'esecuzione di `c.getPrimo()`- versione originale con Security leak**



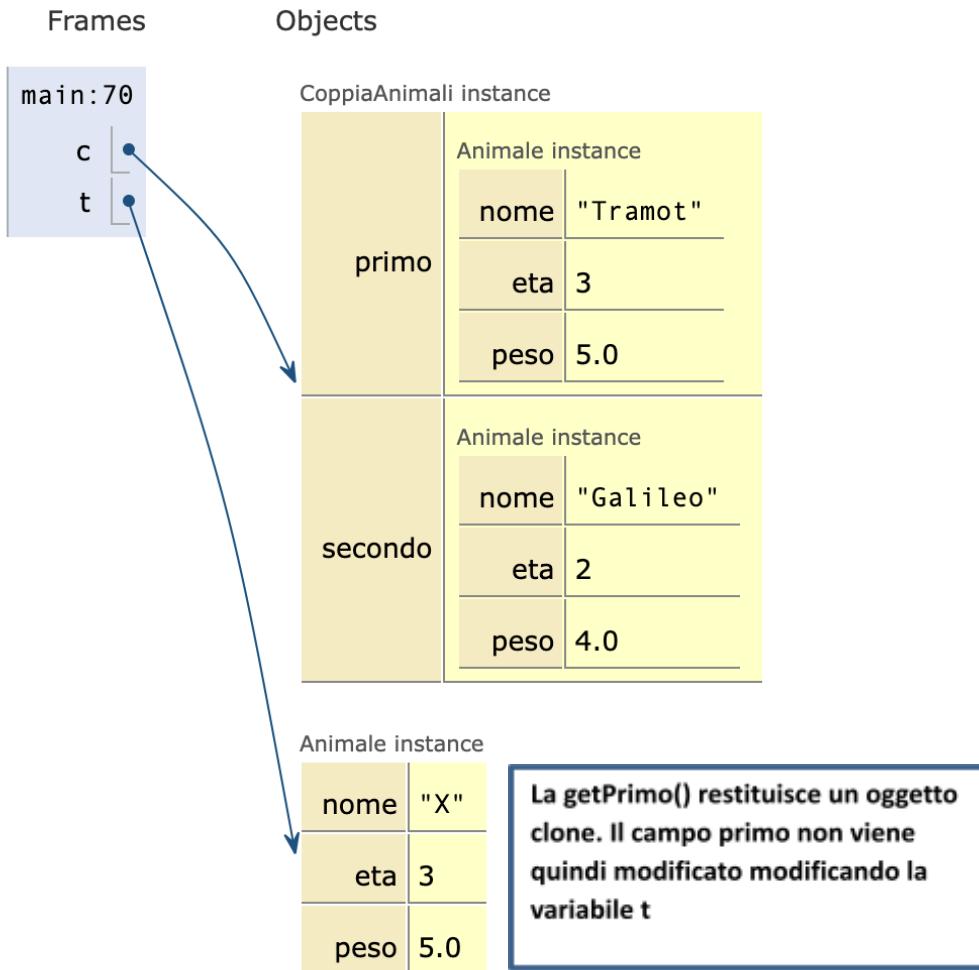
Dopo l'esecuzione di `t.setNome()`



Avendo una copia `t` dell'indirizzo di '`primo`', tramite `getPrimo()` abbiamo potuto modificare il suo campo '`nome`', 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 modificare le `get()` in modo che restituiscano una copia o "clone" dei campi `primo` e `secondo`, con tutti gli attributi uguali all'oggetto originario, ma non l'indirizzo dell'oggetto originario. In questo modo, si permette di conoscere i dati dall'esterno (per esempio, sapere come si chiamano gli animali contenuti in "`c`") ma non di modificare il contenuto di "`c`".

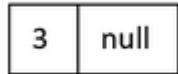
Nel prossimo disegno, vediamo come, modificando le `getPrimo()` e `getSecondo()` di `CoppiaAnimali` in modo che restituiscano dei cloni invece degli indirizzi contenuti nei campi `primo` e `secondo`, si protegge da modifiche l'oggetto originale. Nel codice presentato sopra si possono anche trovare le versioni delle `getPrimo()` e `getSecondo()` che clonano, commentate.



**Lezione 09. Parte 2. Pile dinamiche.** In questa parte introduciamo le liste dinamiche, che godono dell'importante proprietà di non avere una dimensione massima predefinita. Però non lo facciamo con una struttura generale, bensì definendo lo stack, scelta che ha due vantaggi: (1) ha solo due operazioni principali, push e pop; (2) si presta a essere confrontato con l'implementazione tramite array dello stack vista nella Lezione 05. In alternativa, avremmo potuto definire la coda, anch'essa con due operazioni principali per inserire un elemento in fondo alla lista (put) e per estrarre un elemento dalla testa della lista (get). Ma qui ci concentriamo sulle pile.

Le liste dinamiche sono basate sul concetto di ``nodo'', che è la coppia di un dato (nel nostro caso: un intero) e di un puntatore a un (altro) nodo. Per esempio, la seguente figura

mostra un nodo in cui il campo intero contiene il valore 3 e il puntatore all'altro nodo è null.



L'indirizzo dell'oggetto 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 l'insieme di oggetti che contiene null, ogni nodo che punta a null, ogni nodo che punta a un nodo che punta a null, eccetera.

Ecco 3 nodi diversi da null, con elementi 7,5,3. Ciascun nodo contiene, oltre ad un elemento, l'indirizzo del nodo precedentemente creato. Assumendo un inserimento dei dati in testa, come avviene nelle pile, il primo nodo creato contiene (il valore 3 e) l'indirizzo null nel campo next:



Prima di studiare l'implementazione della pila, analizziamo l'implementazione della **classe Node**, che ha come attributi (privati):

- un elemento di tipo int (il dato memorizzato nel nodo),
- un oggetto next di tipo Node destinato a mantenere l'indirizzo del nodo successore nella lista. Essendo un oggetto, il next è un indirizzo/puntatore,
- un costruttore che richiede un elemento e un puntatore per costruire un nodo,
- i metodi get() e set() dei campi di Node.

```
//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;}
}

```

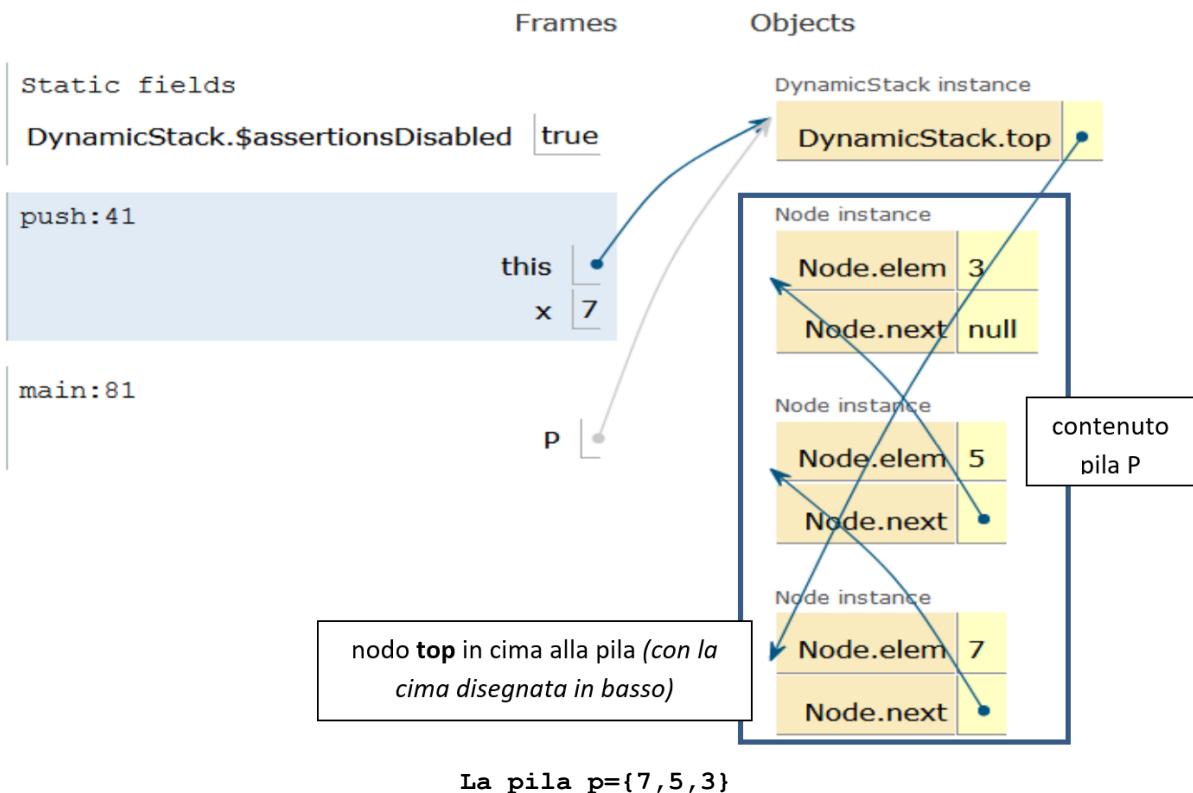
Vediamo ora come utilizzare Node per realizzare **la classe DynamicStack** delle pile dinamiche, che possono contenere un numero arbitrario di elementi e la cui occupazione di memoria cresce o cala in corrispondenza del numero di elementi in esse contenuti. Ricordiamo che nella Lezione 05 avevamo visto la pila statica con struttura interna rappresentata da un array, 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 contiene un elemento e l'indirizzo del nodo successivo, **tranne  $n_{k-1}$  che contiene l'indirizzo null**. L'oggetto pila è implementato con un **unico campo top di tipo Node** (che è quindi l'unico campo della classe), il quale contiene **l'indirizzo del primo nodo**  $n_0$  se  $k>0$ , oppure  $\text{top} = \text{null}$  se  $k=0$  (ovvero se la pila è vuota).

La classe **DynamicStack** delle pile dinamiche ha:

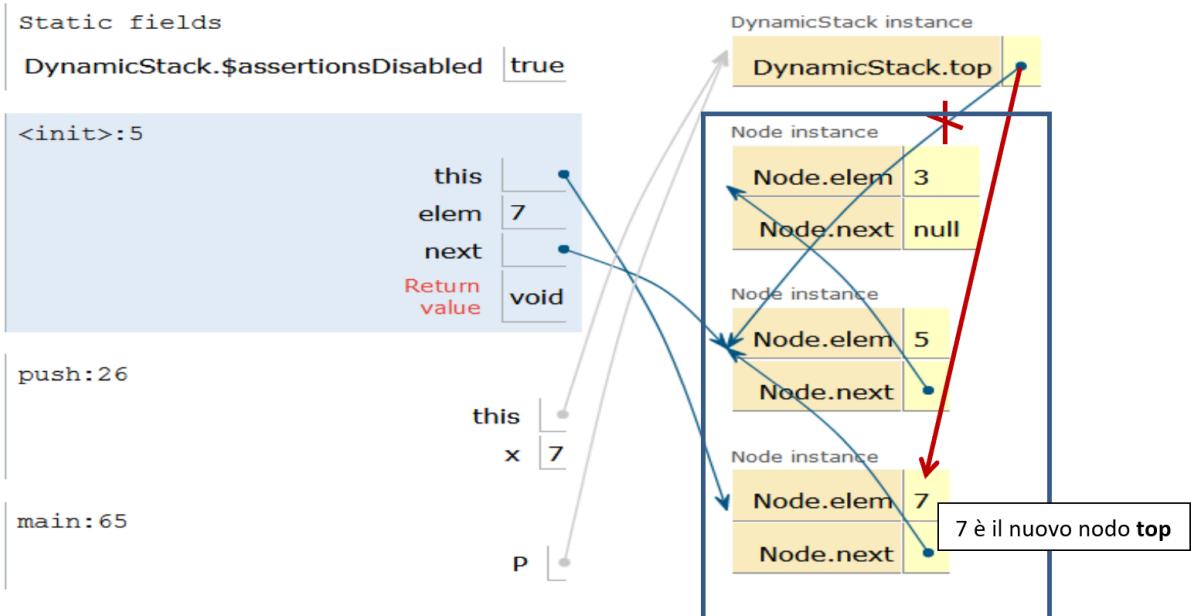
- una operazione **void push(int x)** che aggiunge un nodo di elemento  $x$  "in cima" a una pila (oppure a sinistra se scriviamo la pila da sinistra a destra);
- una operazione **int pop()** che toglie il nodo in cima (o a sinistra) di una pila non vuota e ne restituisce l'elemento;
- **int top()** che restituisce l'elemento del nodo in cima senza eliminarlo;
- è un test **boolean empty()** che controlla se la pila è vuota.
- Implementiamo anche un metodo **toString()** e un costruttore.

Come esempio, vediamo la disposizione nella memoria per la pila **P = {7,5,3}**, qui disegnata con il top che punta al nodo contenente l'elemento **7** raffigurato in basso.



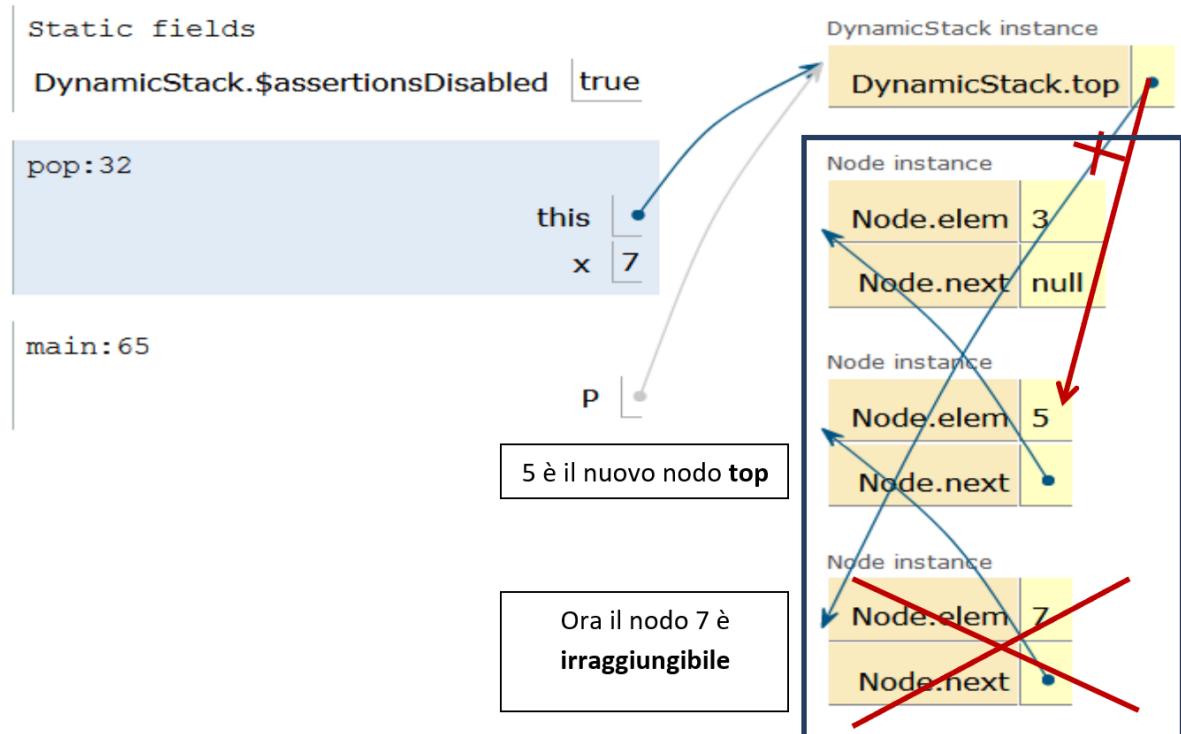
Usiamo la seguente notazione insiemistica per rappresentare nel testo le pile come sequenze ordinate: (i) {} rappresenta la pila vuota; e (ii){a\_i}, con  $0 \leq i <$  "numero elementi nella pila", rappresenta una pila non vuota, dove a\_0 è l'elemento al top della pila e ciascuno degli a\_i è stato inserito come top della pila a un certo momento.

Vediamo ora come è stata ottenuta la pila  $P=\{7, 5, 3\}$  a partire dalla pila  $P=\{5, 3\}$  aggiungendo l'elemento 7 con push(7). Nel disegno, il nuovo nodo contenente 7 punta al nodo contenente 5, che era il vecchio valore di top. Il campo top, che prima puntava al nodo contenente 5, è stato assegnato al nodo contenente 7.



La pila  $P=\{7,5,3\}$  ottenuta dalla pila  $P=\{5,3\}$  con  $\text{push}(7)$

Eliminiamo ora 7 eseguendo il metodo  $\text{pop}()$ : ora top torna a puntare a 5, il nodo 7 non è più raggiungibile e il suo spazio di memoria verrà riciclato dal Garbage Collector.



La pila  $P=\{5,3\}$  ottenuta dalla pila  $P=\{7,5,3\}$  con  $\text{pop}()$

Mostriamo ora il codice della classe **DynamicStack** realizzata con gli oggetti di tipo Node. 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. Si noti infatti che le sue funzionalità dal punto di vista di un codice client che la utilizza (si veda, per esempio, la classe **DynamicStackDemo** più avanti) rimangono le stesse dell'implementazione dello stack con gli array (Lezione 05): cambia l'implementazione interna, che però rimane invisibile al client.

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

    private Node top;
    //punta all'ultimo ultimo nodo aggiunto alla pila, contiene
    null se non ci sono nodi

    /* Tutti i metodi di DynamicStack devono preservare il
    seguente invarianto di classe:
    se la pila è vuota, top contiene null, altrimenti ogni nodo
    tranne il primo punta all'elemento precedente, e il campo top
    di punta al primo elemento della pila.
    */

    //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.getElement();
    return x;
}

/* 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.getElement()+"\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. */
}

//DynamicStackDemo.java (prova della classe DynamicStack)
public class DynamicStackDemo{
public static void main(String[] args){
    DynamicStack p = new DynamicStack();
    p.push(3);p.push(5);p.push(7);p.push(9);p.push(11);
    System.out.println("Stampo la pila p = {11,9,7,5,3}");
    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 estrarre dalla
    pila
    System.out.println(p.top());
    System.out.println("Stampo cosa resta: p = {5,3}");
    System.out.println(p);
}

```

}

**NOTA TECNICA SULL'IMPLEMENTAZIONE DELLA PILA:**

*Si potrebbe pensare di codificare* le pile **direttamente con i nodi, senza introdurre la classe DynamicStack**. In questo caso, la pila vuota sarebbe rappresentata da null. Tuttavia, questa implementazione è problematica:

- dal punto di vista della rappresentazione della conoscenza, creare una pila vuota significa creare comunque un oggetto DynamicStack, che al suo interno non contiene nessun nodo. Infatti, le pile nascono vuote e vengono modificate aggiungendovi gli elementi via via. Non ha quindi senso dire che una pila vuota sia equivalente a null.
- dal punto di vista dello sviluppo delle classi che usano una pila, scegliere questa rappresentazione sarebbe sconveniente. Infatti, chiamare **un metodo su null produce un errore a runtime (NullPointerException)**. Quindi una tale rappresentazione richiederebbe a tutte le classi che utilizzano la pila di gestire il caso della pila vuota come caso particolare (if (pila != null) {...} else {...}).

# Lezione 10

## Metodi statici per la classe Node

**La classe NodeUtil (esercizi su liste concatenate).** In questa lezione proponiamo degli esercizi sui nodi Node (la cui classe è stata introdotta nella Lezione 09) senza considerarli parte di una struttura-dati specifica come DynamicStack, in modo da acquisire disinvoltura nel manipolarli. Questo ci servirà come palestra per comprendere alcune strutture-dati che vedremo più avanti, più complesse dello stack dal punto di vista delle operazioni fornite e/o dal punto di vista della struttura interna.

Chiamiamo **lista concatenata**: (i) [caso base] la lista vuota, in questo contesto rappresentata semplicemente dal puntatore **null**; (ii) [caso induttivo] l'insieme dei nodi raggiungibili da un nodo Node fornito seguendo il puntatore next del nodo, nel caso in cui questo percorso non contenga cicli e termini in un puntatore **null** (anche rappresentato con il simbolo "/").

Queste sono liste concatenate:

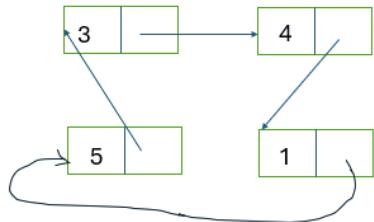
- 1) La lista vuota:

**null**

- 2) Una lista non vuota:



Questa non è una lista concatenata:



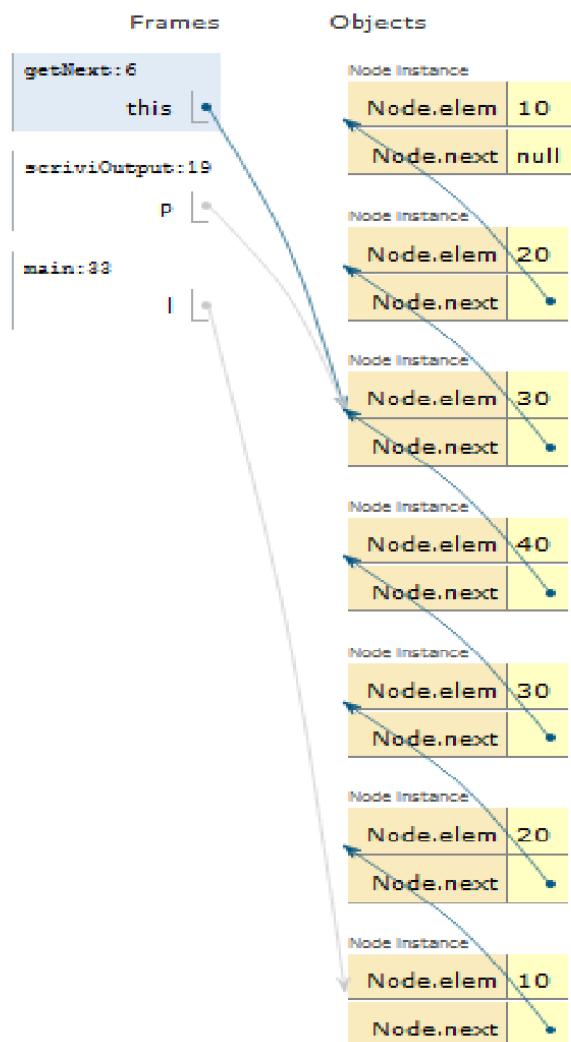
Una lista concatenata si rappresenta quindi con l'oggetto di tipo Node da cui partiamo per percorrerla. Chiamiamo questo

nodo **il nodo in cima** alla lista, oppure: il nodo più a sinistra. Chiamiamo inoltre il nodo a cui punta un nodo dato **il nodo successivo** nella lista.

Definiamo una classe **NodeUtil** con metodi **pubblici e statici** sulle liste concatenate. Le operazioni fornite da questi metodi sono simili dal punto di vista delle funzionalità a operazioni tipiche per la manipolazione di array. Le differenze riguardano l'uso di indirizzi al posto degli 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 (nodo più a sinistra) al posto della posizione **0**;
- (iii) usiamo un Node 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. Vedremo poi un metodo di stampa **scriviOutput()** che scorre la lista l contenente un puntatore a un oggetto di tipo Node. Per non distruggere la lista durante lo scorrimento il metodo usa un puntatore p che parte da l e a ogni passo viene aggiornato a **p.getNext()**.



In questa immagine `scriviOutput()` ha un puntatore `p` al secondo nodo contenente elem uguale 30 (partendo dal basso), ha stampato 30 e sta per passare al nodo successivo con elem uguale a 20 tramite la chiamata a `getNext()`.

Qui riportiamo 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 sia una soluzione iterativa che una ricorsiva. Per i metodi 5-7 non indichiamo suggerimenti. Nel seguito, supponiamo che `q = {10,20,30,40}` sia una lista concatenata.

**0. void scriviOutput(Node p).** Stampa una lista concatenata partendo dal nodo in cima alla lista. Per sviluppare questo metodo, adattate il metodo per stampare una pila della Lezione 08. La chiamata `scriviOutput(q)` stampa: 10 20 30 40, andando a capo dopo ogni elemento.

1. **int length(Node p)**. Calcola la lunghezza di una lista. Attraversiamo la lista con un ciclo, aggiungendo 1 fino a trovare un Node = null. La *length(q)* vale 4.
2. **int sum(Node p)**. Somma degli elementi di una lista. Attraversiamo la lista con un ciclo, sommando tutti gli elementi che incontriamo e mantenendo il risultato in una variabile s. Attraversata tutta la lista, s conterrà la somma di tutti gli elementi della lista. La *sum(q)* vale  $10+20+30+40=100$ .
3. **int max(Node p)**. Restituisce il valore massimo degli elementi di una lista con nodo di partenza diverso da null (ovvero non definito per la lista vuota). Attraversiamo la lista con un ciclo, mantenendo in una variabile m il più grande degli elementi trovati. Alla fine della lista m è il massimo. Il *max(q)* vale 40.
4. **boolean member(Node p, int x)**. Controlla se l'x dato compare in una lista p. Attraversiamo 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. La *member(q, 30)* vale true e la *member(q, 50)* vale false.
5. **String toString(Node p)**. Restituisce una stringa composta concatenando gli elementi dei nodi di p separati da uno spazio. La *toString(q)* vale "10 20 30 40".
6. **boolean sorted(Node p)**. Verifica se una lista concatenata è ordinata in modo debolmente crescente e restituisce true in caso positivo. Il *sorted(q)* vale true. Invece, *sorted(p)* dove  $p = \{10,20,30,40,30,20,10\}$  restituisce false.
7. **boolean equals(Node p, Node q)**. Verifica se due liste concatenate sono uguali: hanno gli stessi elementi nello stesso ordine. L'*equals(q, q)* vale true. Invece, se  $p = \{10,20,30,40,30,20,10\}$ , allora l'*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;}
}

//Applicazione di prova: NodeUtilDemo.java

public class NodeUtilDemo{
    public static void main(String[] args){
        System.out.println( "Main di prova per gli esercizi
0-7");
        System.out.println( "-----");

        //creo una lista concatenata con i nodi di q = {10,20,30,40}:
        //si deve partire da 40 per inserire ogni volta in testa alla
        //lista il nuovo nodo creato

        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:");
        /* Poiché scriviOutput() è un metodo statico di NodeUtil
        dobbiamo invocarlo indicato il nome della classe */
        NodeUtil.scriviOutput(q);
        System.out.println( "-----");

        //creo una lista concatenata con i nodi di
        p={10,20,30,40,30,20,10}: si deve partire da 10

        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:");
NodeUtil.scriviOutput(p);

System.out.println("-----");
System.out.println("1. length(p) = " + NodeUtil.length(p));
System.out.println("1. length_rec(p) = " + NodeUtil.length_rec(p));
System.out.println("-----");
System.out.println("2. sum(p) = " + NodeUtil.sum(p));
System.out.println("2. sum_rec(p) = " + NodeUtil.sum_rec(p));
System.out.println("-----");
System.out.println("3. max(p) = " + NodeUtil.max(p));
System.out.println("3. max_rec(p) = " + NodeUtil.max_rec(p));
System.out.println("-----");
System.out.println("4. member(p,30) = " + NodeUtil.member(p,30));
System.out.println("4. member(p,50) = " + NodeUtil.member(p,50));
System.out.println("4. member_rec(p,30) = " + NodeUtil.member_rec(p,30));
System.out.println("4. member_rec(p,50) = " + NodeUtil.member_rec(p,50));

```

```
System.out.println( "-----" );
System.out.println( "5. toString(q) = "      +
NodeUtil.toString(q));
System.out.println( "5. toString(p) = "      +
NodeUtil.toString(p));
System.out.println( "-----");
System.out.println( "6. sorted(q) = "      +
NodeUtil.sorted(q));
System.out.println( "6. sorted(p) = "      +
NodeUtil.sorted(p));
System.out.println( "-----");
System.out.println( "7. equals(p, q) = "      +
NodeUtil.equals(p,q));
System.out.println( "7. equals(p, p) = "      +
NodeUtil.equals(p,p));
System.out.println( "7. equals(q, q) = "      +
NodeUtil.equals(q,q));
System.out.println( "7. equals(q, p) = "      +
NodeUtil.equals(q,p));
}
}
```

## Soluzioni esercizi 1-7 della Lezione 10

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.** Poiché i metodi di NodeUtil hanno p come parametro formale, essi 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, come invece è servito fare per top nella **toString()** di DynamicStack (nella Lezione 09).

```
//NodeUtil.java
public class NodeUtil{

    //0. Stampa dei nodi di una lista
    public static void scriviOutput(Node p){
        while (p!=null){
            System.out.println(p.getElem());
            p = p.getNext();
        }
    }

    //1. 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 di length()
    public static int length_rec(Node p){
```

```

        if (p==null) return 0;
        else return 1 + length_rec(p.getNext());
    }

//2. 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. Massimo degli elementi di una lista non vuota
//(cioè non uguale a null)
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 già visti, all'inizio m=primo
nodo
    while (p != null){
        // a ogni passo prendo il massimo tra m (max nodi già
visti) e il nodo corrente.
        m = Math.max(m, p.getElement());
        p = p.getNext();
    }
    //alla fine m è 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.getElem(), max_rec(p.getNext()));
    }

//4. 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', quindi si puo restituire false
    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. 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. 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 (elemento successivo < elemento): lista non ordinata
        p = p.getNext();
    }
}

```

```

    //se e` finita la lista, vuol dire non c'e` un elemento >
    del successivo: lista ordinata
    return true;
}

// 7. 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 che sono
        diversi: p, q sono liste 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
}

///////////////////////////////
// USATE IL CODICE APPLICATIVO IN NodeUtilDemo.java //
// per testare questi metodi                         //
/////////////////////////////
}

```

## Lezione 11

### Classi generiche: coppie, nodi e pile

**Lezione 11. Parte 1. Coppie generiche.** Molte costruzioni in Java vengono ripetute uguali per tipi di dati diversi. Alcuni esempi:

- 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. Per esempio:
  - `public class CoppiaIS`  
`{private Integer primo; private String secondo; ...}`
  - `public class CoppiaSD`  
`{private String primo; private Double secondo; ...}`
- Possiamo definire la classe dei nodi il cui contenuto ha tipo T, per un qualunque tipo T. Per esempio:
  - `public class NodeInteger`  
`{private Integer i; private NodeInteger next; ...}`
  - `public class NodeString`  
`{private String s; private NodeString next; ...}`
- Possiamo inoltre definire la classe delle pile e delle code di oggetti di tipo T.

Questo comporta però la moltiplicazione di codice che differisce esclusivamente per i tipi di dati utilizzati nelle classi. Ricordiamo che la programmazione Object Oriented ha come obiettivo il riuso del codice e la riduzione della scrittura di codice, ove possibile. In questo ambito, ha quindi introdotto lo strumento delle classi generiche.

In sintesi, per non ripetere la costruzione del codice per ogni tipo T, S possiamo introdurre **variabili di tipo classe** e costruire una sola volta la classe delle coppie di tipo T, S o la classe dei nodi/pile di tipo T, come classe generica rispetto ai tipi di dati utilizzati. In seguito, nello sviluppo del codice client che utilizza la classe, si sceglieranno i tipi "concreti" i tipi concreti da sostituire a T e S.

**Una classe è detta generica (o parametrica - dove il parametro è un parametro di tipo) se contiene nella sua definizione variabili di tipo.** Questo meccanismo di riuso del codice va sotto il nome di **polimorfismo parametrico**. Ricordiamo che in

Java una classe definisce un tipo. Per capire come usare i generici, vediamo ora un semplice esempio: la 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 angolari: **GenericPair<T,S>**. Per definire una classe generica di coppie, possiamo pensare alla definizione di una qualunque classe *Pair* di coppie concrete di tipi T0, S0 (per es., T0=Integer e S0=Double), e rimpiazzare *Pair* con GenericPair<T,S>, dove T,S sono variabili di tipo. Fa solo **eccezione il costruttore**, che dobbiamo chiamare GenericPair quando lo definiamo, e GenericPair<T,S> quando lo usiamo. Nelle classi generiche si devono anche evitare tutti i metodi che contengono riferimenti a caratteristiche particolari di qualsiasi tipo concreto T0, S0: il codice che resta deve poter funzionare con due classi concrete **qualsunque** che andranno a sostituirsi a T, S. Un esempio per iniziare segue.

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

    public GenericPair(T first, S second) {
        this.first = first; this.second = second;
    }
    // qui i metodi di get e set, etc.
}

public class TestGenericPairOK {
    public static void main(String[] args) {
        GenericPair<Integer, String> p      =      new
GenericPair<Integer, String>(10, "Mario");
/* ho creato un oggetto di tipo GenericPair dove i due tipi
generici vengono istanziati rispettivamente con Integer e
String.*/
    }
}

public class TestGenericPairWrong {
    public static void main(String[] args) {
        GenericPair p = new GenericPair(10, "Mario"); //NO!!
occorre specificare i tipi di dati da utilizzare, sia nella
dichiarazione della variabile che nella new!
```

```

// vd. la spiegazione subito dopo l'esempio
    GenericPair<Integer, String> q = new
GenericPair<String, Boolean>("Mario", true); //NO! Errore di
compilazione: Mismatch di tipo!
//Le istanze di GenericPair<Integer, String> devono avere come
prima componente un oggetto di tipo Integer e come seconda un
oggetto di tipo String (non una String e una Boolean) !
}
}

```

**Linea-guida di good practice sull'uso dei generici.** Può capitare di scrivere **soltanto GenericPair** come tipo di una variabile della classe `GenericPair<T,S>`, omettendo di precisare il valore delle variabili di tipo `T`, `S`, come nell'esempio che segue:

```

public class Test{
    public static void provaS(GenericPair<String, String> p){
        // questo metodo si aspetta un GenericPair composto da due
oggetti String
        System.out.println(p);
        System.out.println();
    }

    public static void main(String[] args){
        GenericPair p6 = new GenericPair("Mario",3);
        // non abbiamo indicato i valori delle variabili di tipo
        provaS(p6);
        // il compilatore accetta questa chiamata a provaS() anche
se il tipo dichiarato di p6 è GenericPair e quello vero è
        GenericPair<String, Integer>, non GenericPair<String, String>
che abbiamo specificato nella firma del metodo provaS()!
    }
}

```

Un tipo generico usato senza istanziare le sue variabili di tipo è detto *raw type*. I raw type sono utilizzati dal compilatore Java per tradurre i generici in bytecode (il linguaggio intermedio in cui il codice Java viene compilato), ma sono anche permessi nel codice sorgente Java. Un raw type nel codice sorgente fornisce però dell'informazione incompleta. Come esempio, vediamo cosa succede con il raw type

GenericPair presente nel nostro codice. Il compilatore, pur compilando, restituisce un warning:

```
vivianabono@MacBook-Pro-9 Materiale % javac Test.java
Note: Test.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Se proviamo a ricompilare con l'opzione -Xlint, come suggerito, otteniamo:

```
vivianabono@MacBook-Pro-9 Materiale % javac -Xlint Test.java
Test.java:39: warning: [rawtypes] found raw type: GenericPair
  GenericPair p6 = new GenericPair("Mario",3);
  ^
missing type arguments for generic class GenericPair<T,S>
where T,S are type-variables:
  T extends Object declared in class GenericPair
  S extends Object declared in class GenericPair
Test.java:39: warning: [rawtypes] found raw type: GenericPair
  GenericPair p6 = new GenericPair("Mario",3);
  ^
missing type arguments for generic class GenericPair<T,S>
where T,S are type-variables:
  T extends Object declared in class GenericPair
  S extends Object declared in class GenericPair
Test.java:39: warning: [unchecked] unchecked call to
GenericPair(T,S) as a member of the raw type GenericPair
  GenericPair p6 = new GenericPair("Mario",3);
  ^
where T,S are type-variables:
  T extends Object declared in class GenericPair
  S extends Object declared in class GenericPair
Test.java:41: warning: [unchecked] unchecked method
invocation: method provaS in class Test is applied to given
types
  provaS(p6);
  ^
required: GenericPair<String,String>
found:  GenericPair
Test.java:41: warning: [unchecked] unchecked conversion
  provaS(p6);
  ^
required: GenericPair<String,String>
found:  GenericPair
```

## 5 warnings

Notate come il compilatore con l'opzione -Xlint abbia trovato tutti i punti di possibile ambiguità data dall'incompletezza dell'informazione fornita nelle istruzioni del main().

In particolare, nella parte evidenziata **in rosso** il compilatore ci fa notare che passiamo un argomento di tipo GenericPair al metodo provaS(), che si aspetta invece un GenericPair<String, String>. Tuttavia, il compilatore non dà errore, solo dei warning, e compila il codice sorgente.

Inoltre quando eseguiamo il bytecode viene stampata la coppia (Mario,3) come da coppia costruita nel main(). In pratica, il tipo dell'argomento del metodo sembra essere ignorato.

Ma perché il compilatore non dà errore invece di fornire solo un warning? **Perché il compilatore rinuncia a applicare i controlli di tipo, non avendo abbastanza informazioni, e delega tali controlli alla JVM** (l'interprete del bytecode), **introducendo quindi possibili errori a runtime**.

La scelta di permettere i raw type nei programmi sorgenti Java è dovuta al voler mantenere funzionante il codice legacy, in particolare quello scritto quando nessuna delle classi delle librerie Java erano generiche (comprese quelle che oggi lo sono), ovvero prima di JDK 5.0. Potrebbe essere interessante capire come il compilatore traduce i tipi generici in bytecode per comprendere meglio questi argomenti, ma non è materia di questo insegnamento. A voi basti sapere che per evitare errori subdoli è buona pratica (anzi, considerate obbligatorio!) istanziare sempre le variabili di tipo. Nel nostro esempio, non utilizzate solo GenericPair come tipo, ma scrivete GenericPair<T0,S0>, dove T0 e S0 sono i tipi concreti che avete scelto, come in GenericPair<Integer,String>, nel codice client.

Ricordiamo che è però consentita **l'abbreviazione GenericPair<>**, detta "diamond notation", che evita di scrivere **nel lato destro dell'assegnamento** dopo il nome del costruttore i tipi già specificati nel sinistro, come in:

```
GenericPair<Integer, String> p = new GenericPair<>(10,  
"Mario");
```

Questa notazione è corretta perché il compilatore è in grado di dedurre in modo unico le istanziazioni dei tipi T e S, cioè i tipi concreti da sostituire a T e S.

Segue il codice completo dell'esempio sulle coppie generiche.

```
//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; }  
  
    /* Ricordiamo che un metodo toString() sta in qualsiasi classe T0, S0 quindi puo' essere usato anche se non sappiamo, all'interno della classe GenericPair, quali saranno i tipi di dati effettivi assegnati nel codice client a T e S. Infatti, se le classi sostituite a T e S non definiscono il toString(), per i campi first e second viene usato quella della classe Object, che restituisce la stringa indirizzo-nella-heap in esadecimale, come NomeClasse@548c4f57. */  
    public String toString()
```

```

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

```

Come esempio di metodo **statico generico** definiamo nella classe TestGenericPair sottostante (quindi fuori dalla classe GenericPair<T,S>) un metodo che, dati i tipi T,S e una coppia di tipo GenericPair<T,S>, inverte le componenti della coppia, e restituisce un risultato di tipo GenericPair<S,T>. Si noti che il metodo costruisce una nuova coppia e non altera quella originaria.

**Se un metodo contiene dei riferimenti a dei tipi generici T, S, e ha come tipo di ritorno la classe generica Classe<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(...)
```

Poiché nel nostro esempio vogliamo definire il metodo all'interno della classe TestGenericPair (quindi fuori da GenericPair<T,S>), **dobbiamo dichiarare che T,S sono variabili di classe prima del tipo di ritorno, scrivendo <T,S>.**

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

    /* Questo metodo, dato un (x,y) di tipo GenericPair<T,S>, restituisce un nuovo (y,x) di tipo GenericPair<S,T>. 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<>, purché la versione
del compilatore Java sia in grado di dedurre in modo unico i
valori di S, T.*/
GenericPair<String, Integer> p = new GenericPair<>("pluto",
1);
// nel seguito, "\t" è la tabulazione e inserisce spazi
bianchi
System.out.println( "p = \t\t\t\t" + p);
System.out.println( "inv(p) = \t\t\t" + inv(p));
System.out.println( "p non cambia: \t" + p);
//Dato che inv è un metodo statico, al di fuori della sua
//classe deve venire chiamato p. es. come:
// TestGenericPair.<String, Integer>inv(p);
}
}

```

**Lezione 11. Parte 2. Nodi generici e tipo Integer.** 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:

```
public class GenericNode<T> {...}
```

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>.

Come già discusso, attenti 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 informazioni essenziali per il programma. È consentita invece **l'abbreviazione GenericNode<>**, purché il compilatore 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 a questo problema, Java ci fornisce le classi Integer, Boolean, Double, detti tipi **wrapper**, fatte da un indirizzo che punta a un oggetto che contiene un intero, booleano e reale, rispettivamente. Immaginiamo l'indirizzo di un oggetto intero, booleano, reale come una **box** ("scatola") che contiene un valore 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 poiché, se necessario, la JVM trasforma automaticamente un int in un Integer e viceversa con operazioni dette **autoboxing (inscatolamento automatico)**. Nello specifico, *boxing* indica l'operazione di mettere, per esempio, un int in un Integer; *unboxing* indica il viceversa.

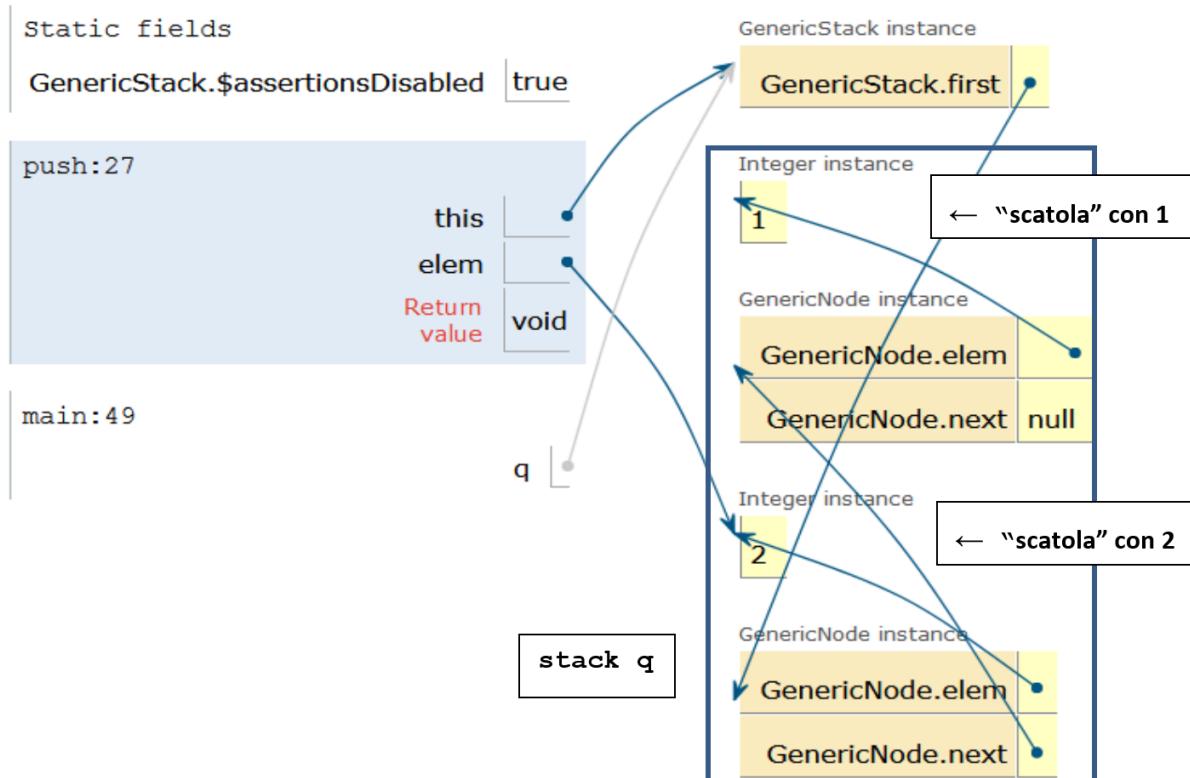


Immagine dello stack `q={2,1}` durante l'esecuzione di `push(2)` (per la descrizione vedi sotto).

Nel disegno qui sopra vediamo come esempio uno stack `q = {2,1}` i cui elementi hanno tipo Integer anziché int. Lo stack `q` ha il campo di tipo Node `first` che punta al primo corrente dei due nodi, che contiene come campo elem l'indirizzo di 2. Si noti che nella figura gli oggetti di tipo Integer vengono indicati con "Integer instance" e non int, in quanto,

appunto, non si tratta di tipi semplici. Si noti anche che elem è disegnato come una freccia che punta a una "scatola" attorno a 2. Il nodo che ha come elem 2 contiene anche l'indirizzo di un nodo che ha come elem 1 ed è fatto allo stesso modo: il campo elem contiene una freccia che punta a 2, non all'int 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>, che descrive 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 con elementi di tipo *int* a partire dalla classe Node dei nodi con elem di tipo *int*.

```
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' il compilatore 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.*;
//in questo modo importiamo anche la classe Random dei
generatori di numeri casuali, oltre alle altre util usate nel
programma

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. */
GenericStack<String> p = new GenericStack<>(); //p pila
String
p.push("world!");

```

```

// OK: essendo p un GenericStack<String>, il metodo push può
ricevere un parametro attuale di tipo String
p.push("hello ");
System.out.println(p); // stampo 2 stringhe
String s1 = p.pop();
// OK: il metodo pop restituisce un valore di tipo String in
quanto p è un GenericStack<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 un valore 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"
// nella heap)

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 restituisce un Integer da cui la JVM
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(), restituendo il contenuto dello
stack in formato String, ci fornisce una versione stampabile
per GenericStack di elementi di tipo arbitrario (ma tutti
dello stesso tipo in quanto, come visto precedentemente, il
compilatore ammette solo la push di elementi del tipo
specificato nella variabile GenericStack<>)
*/
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 12

### Ereditarietà e assert

Ricordiamo che **l'ereditarietà è uno dei principi fondamentali dell'object-oriented programming** (insieme all'information hiding-incapsulamento, che abbiamo già trattato, all'astrazione e alle due forme di polimorfismo, quello parametrico, che abbiamo introdotto nella Lezione 11, e quello per inclusione, detto anche per sottotipo, che tratteremo).

L'ereditarietà **serve a riutilizzare attributi e metodi di classi già definite per concentrarsi sullo sviluppo del "codice nuovo"**.

Il vantaggio di utilizzare l'ereditarietà è innanzitutto il fatto che si evita di scrivere più volte lo stesso codice. Inoltre, quando si modifica il codice ereditato, tutte le classi che erediteranno tale codice modificato utilizzeranno la nuova versione. Si aumenta dunque la sintesi e la manutenibilità del software.

**L'ereditarietà è basata sull'"estensione di classi".**

**Lezione 12. Parte 1. Un primo esempio di estensione di una classe: la classe BottigliaConTappo.** Vediamo come definire una nuova classe D a partire da una classe data C, aggiungendo nuovi attributi e/o nuovi metodi, riscrivendo una parte dei metodi già esistenti in C specializzandoli, e riutilizzando quei metodi di C che non si vogliono cambiare. La classe D viene detta *estensione* o *sottoclasse* di C e diciamo che D *eredita* le componenti di C. Questo meccanismo crea una gerarchia di classi che condividono codice (poiché le sottoclassi ereditano le componenti della loro sopraclassa), promuovendo una forma importante di riuso del software.

Come esempio, riprendiamo la classe Bottiglia della Lezione 06 e supponiamo di voler modellare bottiglie dotate di tappo, che possono essere aperte o chiuse, con la regola che una bottiglia per dare o ricevere acqua deve essere aperta.

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() degli attributi, si hanno metodi per aggiungere e rimuovere una quantità a una bottiglia, nei limiti della sua capienza. Per evitare modifiche alla capacità della bottiglia non forniamo un metodo set() per tale attributo.

```

// 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){
        assert (0 < 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){
        assert quantita >= 0:
        "la quantita' doveva essere >=0 invece vale " + quantita;
        int rimossa = Math.min(quantita, livello);
        livello = livello - rimossa;
        assert (0<=livello) && (livello <= capacita);
        return rimossa;
    }

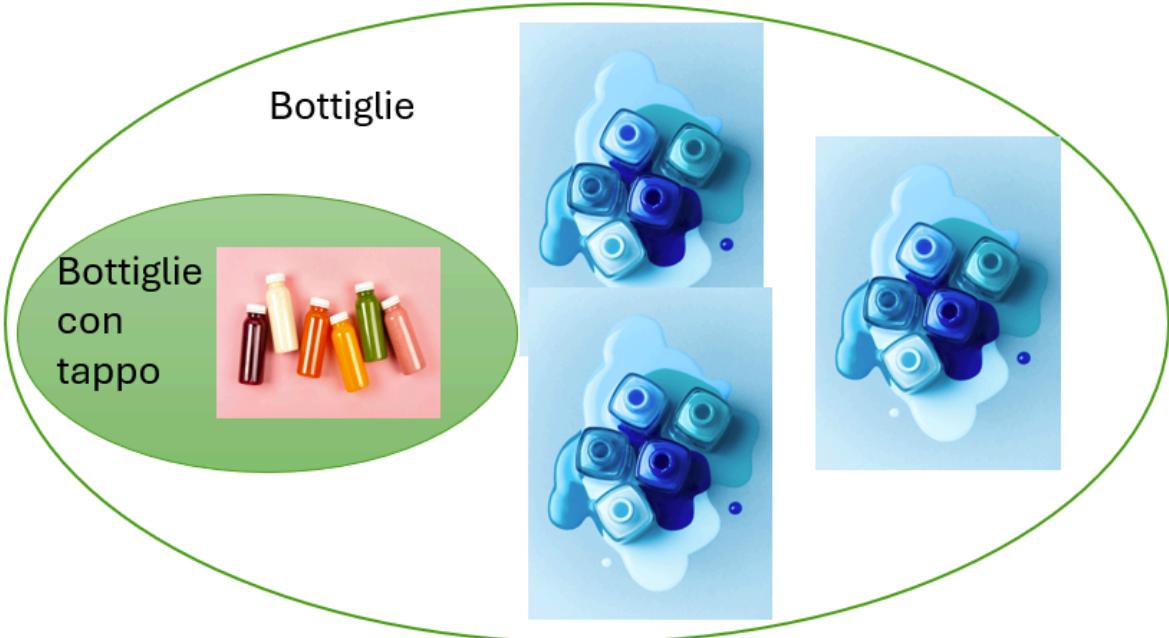
    public int getCapacita(){ return this.capacita; }
    public int getLivello() { return this.livello; }

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

```

**Estensione di una classe.** Se riteniamo importante rappresentare anche delle bottiglie dotate di tappo, l'attuale versione di Bottiglia non è sufficiente perché non ha tale attributo. D'altro canto, non è detto che ogni oggetto di tipo Bottiglia debba avere il tappo. Per risolvere il problema, possiamo quindi definire una nuova classe che ha le stesse caratteristiche di Bottiglia e in più ha un attributo "aperta", rappresentato da una variabile booleana che ci dice se la bottiglia è tappata o meno. La bottiglia può essere in due stati: aperta o chiusa. Vogliamo fare in modo che il versamento di liquido dalla e nella bottiglia abbia effetto solo quando la bottiglia è aperta. Per fare questo, **estendiamo** la classe Bottiglia, creando una **sottoclasse** **BottigliaConTappo**, come descritto nel seguito.

**Estensioni come sottotipi.** Una classe D sottoclasse di una classe C induce un tipo che è un sottoinsieme del tipo indotto dalla sopraclass C (ricordiamo che in Java le classi sono tipi). Questa idea è intuitiva: all'insieme degli oggetti C=Bottiglia appartengono anche gli oggetti della classe D=BottigliaConTappo. In altre parole, gli oggetti in D=BottigliaConTappo sono anche oggetti della classe C=Bottiglia. Altri oggetti di Bottiglia non hanno il tappo e quindi sono di tipo/classe C=Bottiglia ma **non** di tipo/classe D=BottigliaConTappo. Questo implica che un oggetto BottigliaConTappo sia anche un oggetto Bottiglia, quindi un oggetto BottigliaConTappo si può usare ovunque si possa usare un oggetto Bottiglia, ma non vale il viceversa.



**Vediamo con più precisione cosa consente l'estensione.**  
 Partendo da una classe "genitore", detta *sopraclass*, una sottoclasse eredita tutti i membri *public* e *protected* (vedremo tra un po' cosa vuol dire "protected") del suo genitore, indipendentemente dal package in cui si trova la sottoclasse. Se la sottoclasse è nello stesso package del suo genitore, eredita anche i membri *package-private* del genitore (ma non quelli *private*, sempre e solo accessibili alla classe che li definisce). Si possono usare i membri ereditati così come sono, oppure nasconderli (*hiding*), sostituirli o specializzarli (*overriding/sovrascrittura*) con nuovi membri, come elencato nei prossimi punti:

#### CAMP:

- I campi ereditati possono essere usati direttamente, proprio come qualsiasi altro campo.
- Si può dichiarare un campo nella sottoclasse con lo stesso nome di uno della superclasse con qualsiasi visibilità, nascondendo (***hiding***) quello della sopraclass che non sarà più accessibile (**questa pratica è altamente sconsigliata**).
- Si possono dichiarare nuovi campi nella sottoclasse che non sono nella superclasse.

#### METODI:

- I metodi ereditati possono essere usati direttamente così come sono.
- Si può scrivere un nuovo metodo di istanza nella sottoclasse che abbia la stessa firma di quello nella

superclasse, sovrascrivendo **(overriding)** o specializzandolo (overriding in cui il nuovo corpo del metodo nella sottoclasse include chiamate al metodo con la stessa firma della sopraclasse).

- Si può scrivere un nuovo metodo statico nella sottoclasse che abbia la stessa firma di quello nella superclasse, nascondendo **(hiding)** quello della sopraclasse che non sarà più accessibile.
- Si possono dichiarare nuovi metodi nella sottoclasse che non sono nella superclasse.
- Si può scrivere un costruttore di sottoclasse che invoca il costruttore della superclasse, implicitamente oppure usando la parola chiave **super**.

Si noti la **differenza tra "hiding" e "overriding"**: "hiding" implica che, una volta ridefinito nella sottoclasse il membro della superclasse, il membro della superclasse non è più accessibile (nel caso di campo o di metodo statico con la stessa firma). Invece, "overriding" comporta che il membro della superclasse sia ancora accessibile (nel caso di metodi di istanza).

Nel nostro caso, faremo overriding dei metodi aggiungi() e rimuovi() di Bottiglia, per tener conto che nella sottoclasse BottigliaConTappo 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 un'estensione D di C.

Vediamo ora un esempio: BottigliaConTappo estende la classe Bottiglia

```
// 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;
    ...
}
```

Dato **class D extends C**, all'interno della definizione di D, la parola-chiave *super* indica (staticamente: capiremo questo dettaglio più avanti) la classe C che viene estesa da D. Un costruttore della classe estesa D **inizializza sempre con l'invocazione super(...)** a un costruttore della sopraclasse C (i puntini stanno per i parametri del costruttore scelto). Nel caso in cui non la si metta esplicitamente, la JVM tenta di invocare il costruttore di default della classe C, che ha 0 parametri. Se però il programmatore ha implementato nella classe C dei costruttori esplicativi e nessuno di questi ha 0 parametri, il compilatore dà errore perché, come ricordiamo, la definizione di anche solo un costruttore esplicito rende quello di default inaccessibile.

L'obbligatorietà di invocare e eseguire un costruttore della sopraclasse è data dal fatto che la responsabilità di inizializzare i campi della sopraclasse è della sopraclasse stessa.

Un altro uso di *super()* può essere presente in un metodo sovrascritto, nel quale **è possibile** invocare la versione dello stesso metodo della sopraclasse con la sintassi ***super.metodo(...)***.

Segue il codice della classe BottigliaConTappo, sottoclasse di Bottiglia. Nel seguito, indichiamo con NUOVO i metodi che vengono definiti in BottigliaConTappo e non esistono nella superclasse Bottiglia.

```
// 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, di BottigliaConTappo */
    public BottigliaConTappo(int capacita){
        /* invochiamo il costruttore della classe superiore per fare
         * le inizializzazioni della capacita' e del livello: */
        super(capacita);
        // supponiamo che la bottiglia sia inizialmente chiusa:
        aperta = false;
    }
    /* La chiamata al costruttore della classe superiore (super)
     * deve essere la prima istruzione del 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()`). Tuttavia, se la classe superiore ha costruttori esplicativi ma 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; }

// Non sovrascriviamo getCapacita() e getLivello(). Tali metodi sono pubblici, quindi BottigliaConTappo li eredita da Bottiglia.

/* OVERRIDE del metodo "aggiungi()" per versare liquido nella bottiglia: per aggiungere liquido richiediamo che la bottiglia sia aperta. Pertanto, non possiamo ereditare aggiungi() di Bottiglia. 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()" definito 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()". Sovrascriviamo il
toString() di Bottiglia per far restituire tutte le
informazioni della bottiglia, incluso se è aperta o meno. Per
fare questo, 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" lascia
invariati i travasi, e che lo stato "bottiglia chiusa" li
azzerà. */
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 12. Parte 2. "Assert o non assert, questo è il problema!" (di Luca Padovani).** Ripassiamo l'uso dell'assert in Java e vediamo un esercizio 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 debba essere 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**. I numeri negativi non ammettono radice quadrata.

Pertanto, la funzione è definita solo per un sottoinsieme di tutti i possibili valori del parametro `x`: 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++;  
}
```

Nel caso sopra, 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 e, con le asserzioni abilitate, l'esecuzione del metodo viene terminata dalla JVM.

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

Qui invece 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 non è previsto che un'operazione di push su una pila restituisca un risultato. 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() restituisce -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 di questo insegnamento. 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 errore** 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 che riguarda un assert.

## Esercizio 1

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;  
}
```

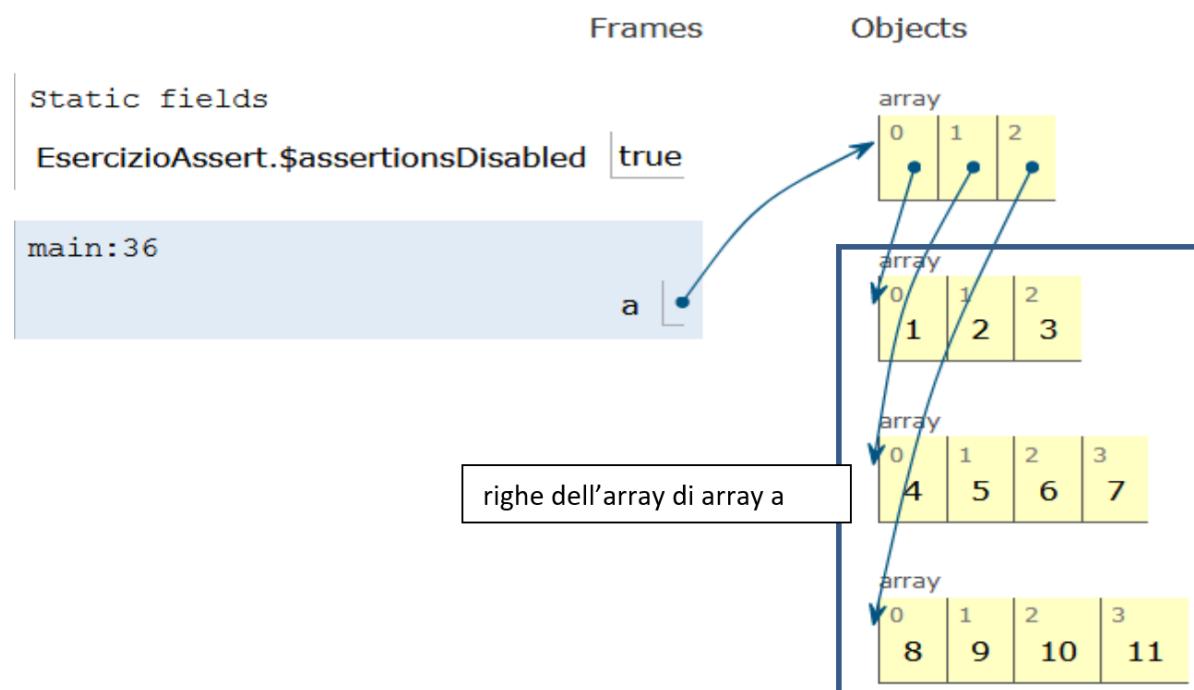
1. Determinare sotto quali condizioni il metodo viene eseguito correttamente (cioè senza generare errori a runtime) e scrivere una corrispondente asserzione da aggiungere come pre-condizione per il metodo. Nello scrivere l'asserzione è possibile fare uso di eventuali metodi statici ausiliari che vanno 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.**

**Prima un breve ripasso sugli array di array.** L'array di array a è l'indirizzo di un array, in cui ogni elemento è, a sua volta, l'indirizzo di un array, che per convenzione possiamo

chiamare *array-riga*. Negli array-riga troviamo degli interi, inoltre gli array-riga non hanno sempre la stessa lunghezza (una struttura di questo genere si dice *matrice ragged*). Per raggiungere l'indirizzo della riga i dobbiamo scrivere `a[i]`, per ottenere l'elemento di posizione j nella riga `a[i]` dobbiamo scrivere `a[i][j]`.

Nel prossimo disegno vediamo un array "a" composto di tre vettori-riga, il primo con {1,2,3}, il secondo con {4,5,6,7}, il terzo con {8,9,10,11}, ovvero:

```
a = {{1,2,3},{4,5,6,7},{8,9,10,11}}
```



## Soluzione dell'Esercizio 1

Si vuole aggiungere in testa al metodo `metodo()` una assert che controlli l'argomento "a" in modo che si evitino errori a runtime nel corpo di `metodo()`. **A tale scopo, scriveremo un metodo statico `ok()` da usare nella assert, di modo che esso restituiscia vero quando l'argomento a di `metodo()` non genera errori a runtime in `metodo()`, falso quanto li genera.** In questo modo, se le asserzioni sono abilitate (`java -ea <nome-file>`) il metodo `metodo()` fallisce a livello dell'asserzione, non durante l'esecuzione del body. Ricordiamo che l'uso delle asserzioni ci serve per ragionare sulle condizioni per evitare errori a runtime dei metodi, fino a che non sapremo usare le eccezioni per gestire tali errori.

1. Supponiamo di chiamare il metodo `metodo()` passando come parametro un array di array che rappresenta una **matrice quadrata**, ovvero in cui:

- gli array-riga  $a[i]$ ,  $0 \leq i \leq a.length$ , hanno tutti la stessa lunghezza (tutte le  $a[i].length$  sono uguali tra loro), ovvero tutte le righe hanno lo stesso numero di elementi, per cui ha senso parlare di "colonne della matrice";
- il numero di righe e il numero di colonne sono uguali ( $a.length = a[i]$ ,  $0 \leq i \leq a.length$ )

in questo caso, `metodo()` non genera errori a runtime e stabilisce se la matrice è simmetrica oppure no rispetto alla diagonale principale. Per esempio:

|   |   |   |   |
|---|---|---|---|
| 1 | 5 | 0 | 6 |
| 5 | 2 | 4 | 7 |
| 0 | 4 | 3 | 2 |
| 6 | 7 | 2 | 4 |

In realtà, per evitare errori a runtime è sufficiente che il numero di elementi in ciascun array-riga non sia minore del numero delle righe. Ovvero, se dato  $r = \text{numero delle righe} = a.length$  e se per ogni array-riga  $a[i]$ , con  $0 \leq i \leq a.length$ , si ha che  $a[i].length \geq r$ , allora `metodo()` non genera errori a runtime.

Non dimentichiamo però di testare anche le seguenti condizioni:  $a$  deve essere diverso da `null` e ogni  $a[i]$ , con  $0 \leq i \leq a.length$ , deve essere diverso da `null`.

Il metodo di controllo `ok()` dell'asset può quindi essere implementato nel seguente modo:

```
public static boolean ok(int[][] a) {
    if (a==null) return false;
    //se a=null allora a.length produrrebbe 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 produrrebbe
    NullPointerException
    //se a[i]<r allora a[i][r-1] produrrebbe
    ArrayOutOfBoundsException
    return true; //se nulla di cui sopra capita: ok(a)=true
}

```

2. Se il metodo ok() non fallisce, allora il metodo metodo() restituisce true se la (sotto-)matrice  $r \times r$ , dove  $r = a.length$ , è simmetrica. Per esempio:

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 5 | 0 | 6 |   |
| 5 | 2 | 4 | 7 | 5 |
| 0 | 4 | 3 | 2 |   |
| 6 | 7 | 2 | 4 |   |

Riportiamo qui tutto il codice.

```

//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) Il 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' = a.length. (2) Altrimenti metodo(a)= true se e solo se la matrice r x r nel lato sinistro di a e' simmetrica.

Nel caso dell'esempio, il metodo ok() impedirebbe l'esecuzione del metodo metodo(), perché r=3 e la seconda riga è solo lunga 2. (a[1]=2<3) \*/

```
//Definiamo il metodo ausiliario ok(a) usato nell'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 13

### Estensioni ripetute di classi

**Lezione 13. Una gerarchia più profonda.** Nello sviluppo di librerie software, può capitare che una classe debba essere ripetutamente estesa per ottenere una gerarchia di sottoclassi via via più specifiche. L'obiettivo, come sempre, è massimizzare il riutilizzo del codice (e quindi la sua manutenibilità). Per esempio, consideriamo la seguente gerarchia IS-A:

**Vehicle:** rappresenta il concetto generico di veicolo;

- Attributi: speed, capacity, fuelType;
- Metodi: start(), stop(), refuel().

**Car:** un tipo specifico di veicolo (sottoclasse di Vehicle); eredita tutti gli attributi e i metodi di Vehicle.

- Attributi aggiuntivi: numberOfWorkers, trunkCapacity (trunk è il bagagliaio);
- Metodi: metodi aggiuntivi: openTrunk(), closeTrunk().

**ElectricCar:** un tipo specifico di automobile (sottoclasse di Car); eredita tutti gli attributi e i metodi di Car (e di Vehicle).

- Attributi aggiuntivi: batteryCapacity, chargingTime;
- Metodi aggiuntivi: chargeBattery();

Perché ci serve avere tre livelli di gerarchia? **Per definire una sola volta i metodi comuni delle sottoclassi.** Per esempio, Vehicle rappresenta tutto ciò che può essere ereditato da Car e da altri tipi di veicolo qui non rappresentati, come i camion (Truck). Inoltre, Car è utile a raccogliere le specificità delle auto, diverse da quelle dei camion. Infine, ElectricCar specifica dettagli dei veicoli elettrici, diversi da quelle dei veicoli a benzina, diesel, gas, etc., qui non rappresentati. Potrebbe essere che qualche classe, come nel caso di BottigliaConTappo, sovrascriva qualche metodo per specializzarlo; però, tutto il resto viene ereditato!

Con questa idea in mente affrontiamo in dettaglio un esempio. Partiamo dalla classe **DynamicStack** delle pile dinamiche (Lezione 09) e supponiamo che ci serva avere una versione delle pile dinamiche che restituisca il valore massimo contenuto al loro interno. Dunque, creiamo una nuova classe che estende DynamicStack e aggiunge un attributo **max** (massimo

valore) con relativo metodo di get. Supponiamo poi che ci serva anche una versione delle pile che, oltre al max, possa indicare quanti elementi contiene la pila. Allora, estenderemo ulteriormente la sottoclass e aggiungendo un attributo **size** (numero degli elementi) e relativo metodo di get.

Estendere comporta aggiungere un'invariante di classe con delle condizioni che descrivano quali valori sono accettabili nei nuovi campi.

```
// classi Node.java e DynamicStack.java: dalla Lezione 09
// nella definizione di DynamicStack della Lezione 09
sostituiamo private Node top con: protected Node top.
Spiegheremo dopo il perche' di questa modifica.
```

```
public class DynamicStack {
    protected Node top;
    /* Rispetto alla classe DynamicStack vista precedentemente,
    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 (si veda più avanti per
    le motivazioni) */
}

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 {
    // Ereditiamo 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, altrimenti, se lo
    stack e' vuoto, il valore di max e' arbitrario */

    // COSTRUTTORE
    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, infatti, 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. Verrà utilizzato nella
pop(). Perché allora questo codice non lo mettiamo
direttamente nella pop()? Perché violerebbe il principio di
buona progettazione High Cohesion, che dice, fra le altre
cose, che ogni metodo deve, se possibile, svolgere un solo
task ben preciso e quello della pop() è di restituire
l'elemento al top dello stack e cancellarlo, non di
ricalcolare max.
// NOTA: possiamo usare il nodo top della classe DynamicStack
perche' abbiamo dichiarato top "protected" e quindi
accessibile nelle classi che estendono DynamicStack (si veda
più avanti per le motivazioni).
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 Node con valore
        l'indirizzo del nodo puntato da 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 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;
}

//EREDITARIETA' - 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";
}
}

```

Come si può vedere nel codice, nella classe DynamicStack.java abbiamo sostituito private con **protected** come modificatore di visibilità del campo top. Questo modificatore di visibilità consente di:

1. utilizzare top in ogni sottoclasse che estende la classe data (nella stessa cartella/package, ma anche in altre cartelle/package);
2. utilizzare top in ogni classe contenuta nella stessa cartella/package.

ma non di utilizzarlo in qualsiasi altro contesto.

Nel nostro caso vogliamo accedere alla posizione top di una pila quando ne calcoliamo il massimo in DynamicStackMax, che è sottoclasse di DynamicStack. Questa visibilità rende più facile la definizione di sottoclassi ovunque siano definite, nello stesso package o in altri package, ma ci impedisce di modificare un attributo di una classe di una cartella/package che importiamo (azione che potrebbe facilmente portare a errori).

Una domanda che ci possiamo fare è se il punto 2. non violi il principio dell'*information hiding*, siccome estendiamo la visibilità dell'attributo all'intera cartella/package a cui appartiene la classe che definisce l'attributo stesso. La risposta è no, se immaginiamo che il codice di un software sia progettato secondo il principio **Low Coupling/High Cohesion**, ovvero che tra classi di package diversi ci sia il minimo necessario di interazione per raggiungere gli obiettivi del software (no interazioni non necessarie), ma che ciascun

package risponda con dei task precisi a obiettivi precisi, per cui è safe che le classi all'interno di uno stesso package possano interagire più agevolmente tra loro.

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
    public DynamicStackSize() {
        super();
        //Invoco il costruttore della classe superiore:0 argomenti
        size = 0;
    }

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

    // OVERRIDE del metodo push(): inserimento elemento in cima
    // alla pila
    public void push(int n) {
        super.push(n); //invoco il metodo push() 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";
}
}

```

Proviamo ora la classe DynamicStackSize, che è estensione della classe DynamicStackMax, a sua volta estensione della classe DynamicStack.

```

//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);

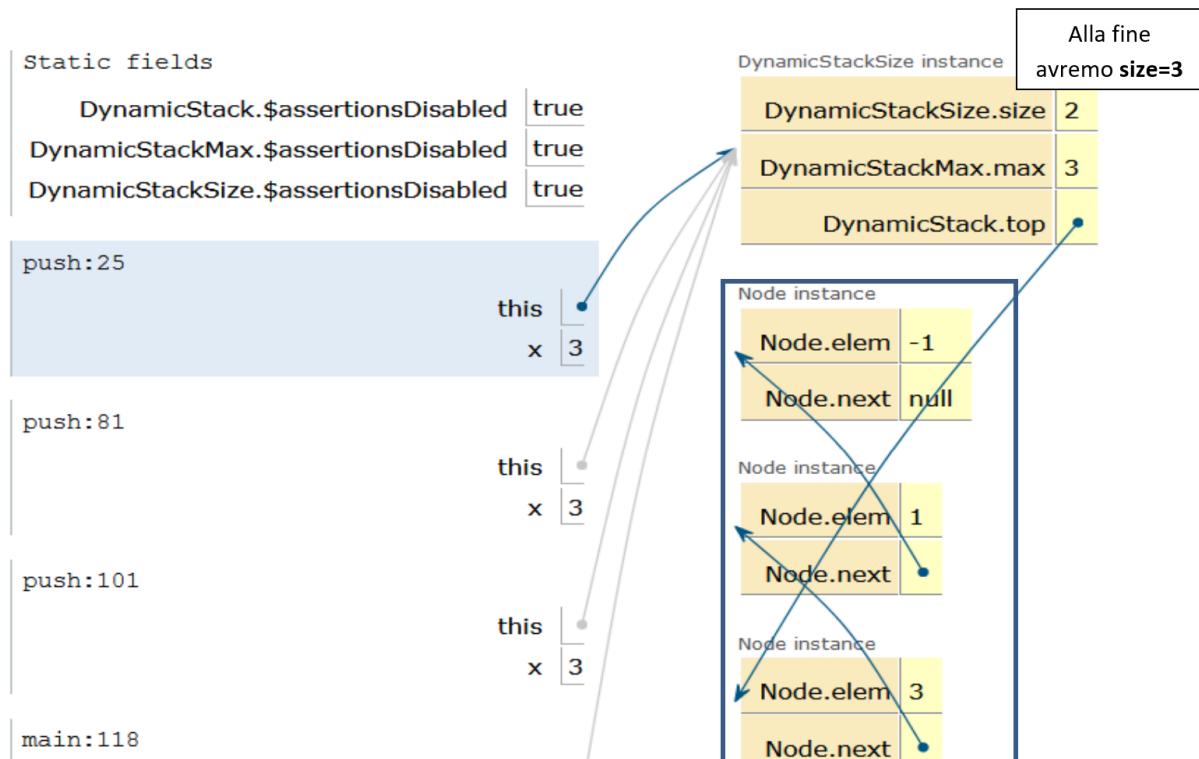
        System.out.println( "Definisco la pila p =
{11,9,7,5,3,1,-1}" );
        p.push(1); p.push(3); p.push(5); p.push(7); p.push(9);
        p.push(11);
        System.out.print(p);

        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 con top():
        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 un oggetto p di tipo DynamicStackSize

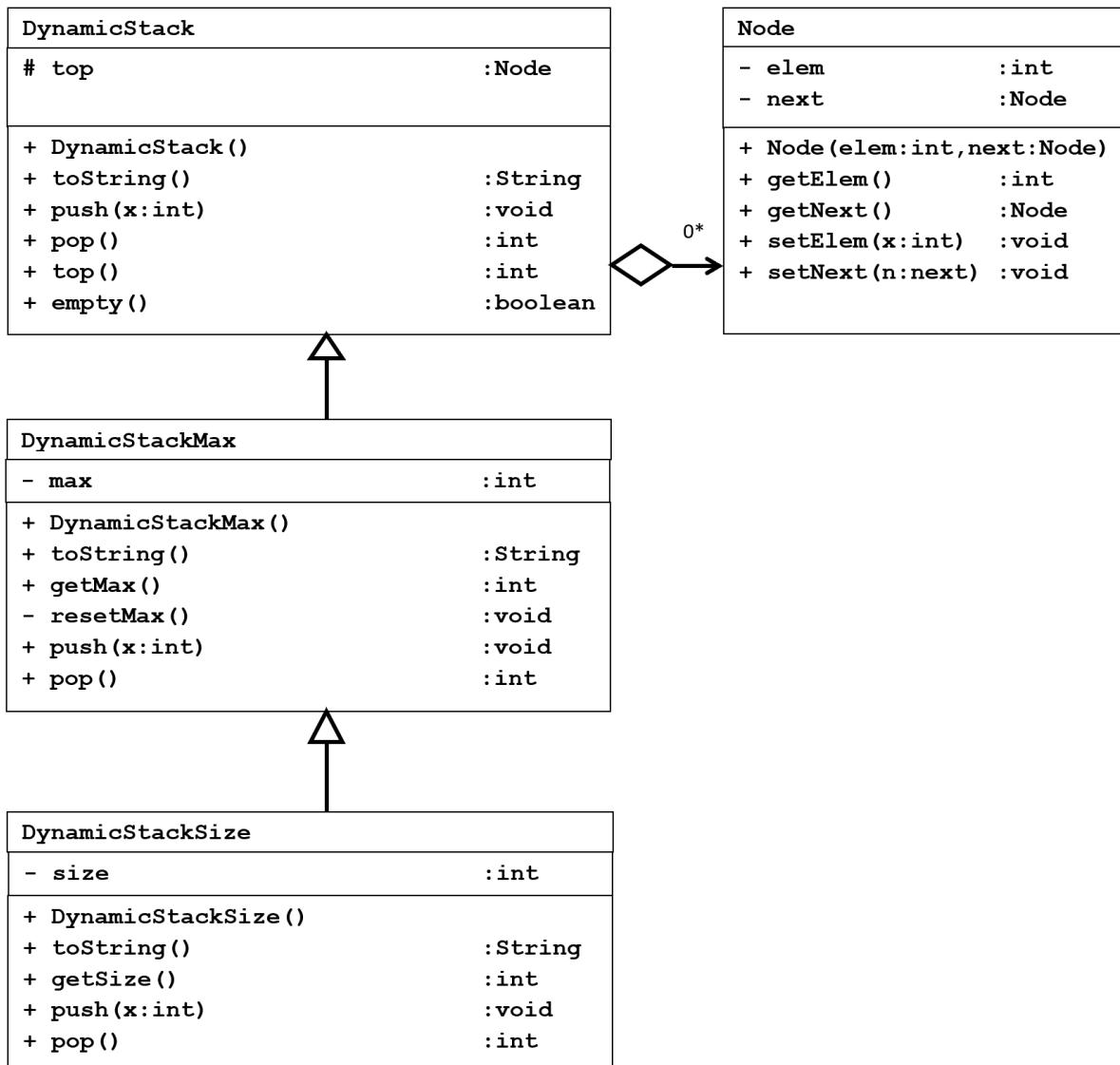
Partiamo da `p={1,-1}` (che rappresenta uno stack con due soli elementi, e con top 1) e eseguiamo `p.push(3)`. Ecco cosa succede. La pila ha tre attributi, l'indirizzo dell'elemento in cima alla pila `top` che fa parte della classe `DynamicStack`, e i due attributi `max` e `size` aggiunti con le due estensioni. Il metodo `push(3)` della classe `DynamicStackSize` richiama il metodo `push(3)` della classe `DynamicStackMax`, che aggiorna `max` e richiama il metodo `push(3)` della classe `DynamicStack`. Alla fine la prima `push(3)`, ovvero la versione della classe `DynamicStackSize`, concluderà l'operazione aggiornando `size`. Nel prossimo disegno vediamo la situazione dello Stack + Heap in questo istante.



Nella classe `DynamicStackSize`, `p.push(3)` richiama altre due `p.push(3)`, quelle delle sopraclassi, seguendo l'ordine gerarchico: prima quella di `DynamicStackMax`, poi quella di `DynamicStack`.

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



# Lezione 14

## Tipo esatto e binding dinamico

**Lezione 14. Parte 1. Tipo apparente e tipo vero. Binding dinamico. Upcast e downcast.**

**Tipo apparente e tipo vero. Binding dinamico.** Si parla di **tipo apparente** per indicare il tipo 'tip' con cui una variabile 'v' viene dichiarata: 'tip v'. Si chiama **tipo vero (o esatto)** di 'v' il tipo a runtime dell'espressione 'expr' che sta a destra in un assegnamento 'v = expr'. Nel caso delle variabili di tipo classe, il tipo vero è quello indotto dalla 'new', quando l'oggetto viene costruito a runtime. Per esempio:

```
(1) BottigliaConTappo b1 = new BottigliaConTappo(10);  
  
(2) Bottiglia b2 = new BottigliaConTappo(10);  
     // assegnamento corretto siccome BottigliaConTappo è  
     sottoinsieme di Bottiglia (si veda nella Lezione 12 e nel  
     seguito)
```

in (1) tipo apparente e tipo vero coincidono (sono tutti e due BottigliaConTappo), mentre in (2) il tipo apparente è Bottiglia, mentre il tipo vero è BottigliaConTappo. Si noti che in questi esempi è immediato distinguere tra tipo apparente e tipo vero, ma in altri contesti non lo è, per esempio nel caso in cui a destra dell'assegnamento ci fosse una chiamata di metodo, magari all'interno di un'espressione composita che usa il risultato del metodo stesso.

**Il compilatore fa i controlli di tipo sui tipi apparenti** che essendo dichiarati nel codice sono a disposizione a tempo di compilazione. **Tuttavia, durante l'esecuzione del programma, a runtime, la versione del metodo da invocare dipende dal tipo vero.** Per esempio, se scriviamo:

```
Bottiglia b1 = new Bottiglia(10);  
int x = b1.aggiungi(3);
```

la versione del metodo aggiungi() che verrà invocata sarà quella della classe Bottiglia, avendo b1 tipo Bottiglia.

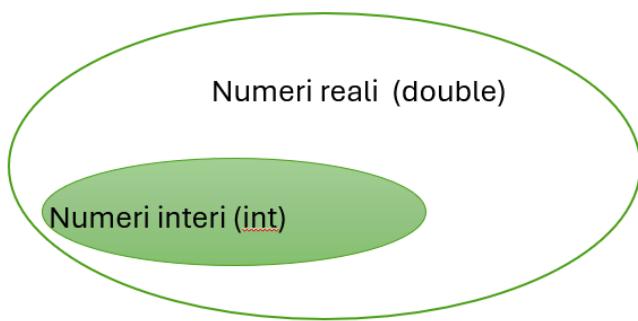
Invece, se scriviamo

```
Bottiglia b2 = new BottigliaConTappo(10);  
int q = b2.aggiungi(3);
```

la versione del metodo aggiungi() che verrà invocata sarà quella della sottoclasse BottigliaConTappo (il tipo vero di b2), che sovrascrive (fa l'override del) metodo corrispondente nella sopraclassse Bottiglia, specializzandolo. Quindi, l'effetto di b2.aggiungi() è che l'aggiunta non avviene se la bottiglia è chiusa, mentre b1.aggiungi() non considera la condizione di apertura o chiusura della bottiglia.

Generalizzando, possiamo dire che, se il programma contenente una chiamata obj.m() compila correttamente, c'è almeno una versione del metodo m() applicabile a obj, quella contenuta nel tipo apparente di obj. La JVM, durante l'esecuzione, dedurrà il tipo vero dell'oggetto obj e lo utilizzerà per **decidere quale versione del metodo m() applicare all'oggetto obj (ovvero da quale classe nella gerarchia di ereditarietà prendiamo il metodo)**. Questo meccanismo viene chiamato **dynamic binding** perché avviene a **runtime**. Talvolta viene anche chiamato **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.

**Sul sottotipo e sui cast (tipi primitivi).** Vediamo ora cosa si intende per **relazione di sottotipo** tra tipi primitivi tramite un esempio. 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 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:

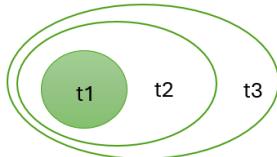
int <: double

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

Dati i tipi t, t1, t2, t3, la relazione di sottotipo è:

- riflessiva:  $t \leq t$ ;
- antisimmetrica: se  $t_1 \leq t_2$  e  $t_1$  diverso da  $t_2$ , allora  $t_2 \not\leq t_1$ ;
- transitiva: se  $t_1 \leq t_2$  e  $t_2 \leq t_3$ , allora  $t_1 \leq t_3$ .

Ovvero  $\leq$  è una relazione d'ordine parziale.



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;
```

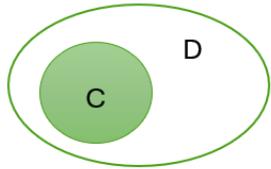
**NON è corretto**, perché si avrebbe perdita di informazione: **si perderebbe la parte decimale di k**. Tuttavia, esistono situazioni in cui si vuole "stringere" un tipo, essendo quindi consapevoli dell'operazione che si sta facendo. Pertanto, Java permette questa perdita di informazione se il programmatore 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;
```

L'effetto è che in y ci sarà solo la parte intera di k. Avremo quindi un **troncamento della parte decimale, non un arrotondamento all'intero più vicino** (per arrotondare, si può, per esempio, utilizzare il metodo di libreria `Math.round()`).

**Sul sottotipo e sui cast (tipi classe)**. Sappiamo già 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. Inoltre, quando si definisce una sottoclasse con "extends" si introduce anche una **relazione di sottotipo**. Supponiamo di avere **class C extends D**. Allora, ogni oggetto di tipo C può essere visto anche come un oggetto di tipo D, ovvero C è un sottotipo di D: **C <: D**



Intuitivamente e similmente al caso di `<`: dei tipi primitivi, questa relazione dice che **ogni oggetto di tipo C si può utilizzare in qualsiasi contesto in cui ci si aspetterebbe un oggetto di tipo D**. Grazie alla relazione di sottotipo, un oggetto di Java può, quindi, avere diversi tipi. Abbiamo perciò un'altra forma di polimorfismo oltre al polimorfismo parametrico (quello dei generici introdotti nella Lezione 11), chiamato **polimorfismo per sottotipo o per inclusione**.

Consideriamo ora il caso specifico:

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

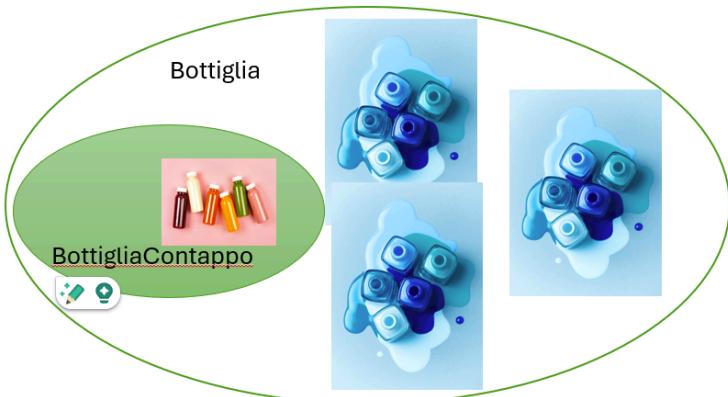
che implica, con la notazione vista sopra, che:

```
BottigliaConTappo <: Bottiglia
```

Anche tra tipi-classe possiamo fare l'**upcast**, ovvero possiamo scrivere la seguente istruzione:

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

che assegna un oggetto di un tipo più piccolo (più specifico) `BottigliaConTappo` a una variabile con tipo più grande (meno specifico) `Bottiglia`. **L'upcast è sempre (matematicamente) corretto perché supportato dalla relazione di sottotipo**: in questo esempio, tutti gli oggetti che appartengono all'insieme `BottigliaConTappo` appartengono anche all'insieme `Bottiglia`.



Se invece scriviamo:

```
BottigliaContappo b3 = new Bottiglia(4);
//ERRORE a tempo di compilazione!
```

abbiamo un errore a tempo di compilazione, perché un oggetto 'o' di tipo Bottiglia non può essere usato in tutti i contesti che si aspettano un oggetto di tipo BottigliaConTappo, per esempio su 'o' non si può chiamare il metodo apri().

Cosa intendiamo per contesto che si aspetta un oggetto di tipo BottigliaConTappo? Pensate a un sistema di automazione in cui un robot inscatola le bottiglie piene di liquido. Se si assume che le bottiglie abbiano il tappo (e siano state precedentemente tappate), andrà tutto bene. Diversamente, si verserà il loro contenuto per terra! L'automazione deve quindi assumere che, in fase di inscatolamento, le bottiglie siano tappate e non può lavorare con oggetti di tipo Bottiglia perché non garantiscono questa proprietà.

Anche tra tipi-classe è possibile fare **downcast** (sempre) esplicativi:

```
//BottigliaConTappo b3 = new Bottiglia(4);
//NON compila perché BottigliaConTappo <: Bottiglia e non
viceversa!
```

```
BottigliaConTappo b3 = (BottigliaConTappo) new Bottiglia(4);
//downcast: COMPIGA!
```

In questo assegnamento si dice al compilatore che l'oggetto creato deve essere considerato di tipo BottigliaConTappo, a prescindere dal tipo vero.

**È importante notare che il tipo vero dell'oggetto appena creato non cambia come conseguenza del cast:** rimane Bottiglia. Quindi si potrebbero avere degli errori a runtime nella parte

di programma che segue quel downcast, per esempio se invocasse il metodo apri() su b3, non contenuto nella classe Bottiglia.

Vediamo ora un esempio in cui, nonostante la presenza di un downcast, non si introducono errori a runtime:

```
BottigliaConTappo c1 = new BottigliaConTappo(3);
Bottiglia c2 = c1; // upcast; c2 alias di c1.
BottigliaConTappo c3 = (BottigliaConTappo)c2; // downcast
```

Il terzo assegnamento dice al compilatore di considerare l'oggetto in c2 (di tipo apparente Bottiglia) come se fosse un oggetto di tipo BottigliaConTappo. Cosa succede a runtime se invochiamo il metodo apri() su c3? Va tutto bene, perché il tipo vero di c2 (e quindi di c3) è BottigliaConTappo. Invece ci sarebbe un errore a runtime se il tipo vero di c3 NON fosse BottigliaConTappo, o un tipo indotto da una sua sottoclasse. Si può anche osservare che c1.apri() compilerebbe perché c1 ha tipo apparente BottigliaConTappo. Invece, c2.apri() non compilerebbe perché c2 ha tipo apparente Bottiglia e Bottiglia non possiede il metodo apri().

L'ultimo esempio può sembrare apparentemente artificiale, ma era voluto per sperimentare su una quantità limitatissima di codice e già contiene quasi tutto quello che serve per capire la relazione di sottotipo e i cast. Facciamo ora un esempio più esteso: si definiscono due classi, la prima è Lib, che contiene un metodo statico prova() con parametro formale di tipo Bottiglia e la seconda è LibDemo, che contiene un main() il quale usa il metodo Lib.prova(). La descrizione e le relative spiegazioni del comportamento si trovano nel codice stesso:

```
//Lib.java
public class Lib {
    public static void prova(Bottiglia b) {
        BottigliaConTappo c = (BottigliaConTappo)b;
        // si deve fare questo cast e non usare b direttamente
        sennò si riceve il seguente errore di compilazione. Infatti,
        il tipo apparente di b è Bottiglia che non contiene apri():
        /*
            error: cannot find symbol
            b.apri();
            ^
            symbol:   method apri()
            location: variable b of type Bottiglia
}
```

```

        */
        c.apri();
        int liv = c.aggiungi(1);
    }
}

// LibDemo.java
public class LibDemo{

    public static void main(String[] args){

        System.out.println("## b1 ##");
        BottigliaConTappo b1 = new BottigliaConTappo(9);
        System.out.println("## prima ##");
        System.out.println(b1);
        Lib.prova(b1);
        System.out.println("## dopo ##");
        System.out.println(b1);
        //Upcast: BottigliaConTappo <: Bottiglia, dove Bottiglia è
        il tipo apparente del parametro b di Lib.prova(). Compila e
        non dà errori a runtime in Lib.prova(), perché ha tipo vero
        BottigliaConTappo.

        System.out.println("## b2 ##");
        Bottiglia b2 = new Bottiglia(2);
        System.out.println("## prima ##");
        System.out.println(b2);
        Lib.prova(b2);
        System.out.println("## dopo ##");
        System.out.println(b2);
        //Compila perché b2 ha tipo apparente Bottiglia, che è
        proprio il tipo del parametro di Lib.prova(), MA si ha il
        seguente errore a runtime nel metodo Lib.prova(),
        all'istruzione 'BottigliaConTappo c = (BottigliaConTappo)b;';
        /*
        Exception in thread "main" java.lang.ClassCastException:
        class Bottiglia cannot be cast to class BottigliaConTappo
        (Bottiglia and BottigliaConTappo are in unnamed module of
        loader 'app')
        at Lib.prova(Lib.java:4)
        at LibDemo.main(LibDemo.java:10)
        */
    }
}

```

Questo esempio è interessante perché il codice client di Lib e, in particolare di Lib.prova(), sta in un'altra classe (LibDemo). In generale, **i metodi di una libreria non possono prevedere come verranno usati da un qualsiasi codice client, quindi vanno scritti in modo robusto.**

Se vogliamo che il metodo possa lavorare su oggetti di tipo Bottiglia (con o senza tappo): per rendere più robusto il metodo Lib.prova() possiamo scriverlo utilizzando il predicato **instanceof** così definito:

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

L'**instanceof** controlla se il tipo vero di obj corrisponde a C, escludendo il caso in cui obj è null.

Riscriviamo ora una versione più robusta del metodo Lib.prova():

```
public static void prova(Bottiglia b) {  
    int liv;  
    if (b instanceof BottigliaConTappo){  
        BottigliaConTappo c = (BottigliaConTappo)b;  
        c.apri();  
        liv = c.aggiungi(1);  
    }  
    else  
        liv = b.aggiungi(1);  
  
}
```

In questo modo, il downcast che precede la chiamata al metodo apri() viene eseguito solo se b ha tipo vero BottigliaConTappo e si evita l'errore a runtime java.lang.ClassCastException se b ha tipo vero Bottiglia.

Ora facciamo qualche altro esempio per sperimentare quanto visto fino ad ora. Ricordiamo che:

- **il compilatore fa i controlli di tipo usando i tipi apparenti;**

- a runtime, viene scelta la versione di un metodo (non-statico) in base al tipo vero dell'oggetto su cui viene invocato il metodo.

### Esempio 1

```
// Bottiglia.java. Riutilizziamo il programma della Lezione 06

// BottigliaConTappo.java. Riutilizziamo il programma della
Lezione 12

// TestCast.java
public class TestCast {
    public static void main(String[] args){
        // ESEMPIO. A seconda se la variabile 'oggi_piove' è
        // 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: passaggio a una classe superiore, che induce un
        // tipo più grande (meno preciso). E' sempre corretto.
        Bottiglia b;
        if (oggi_piove) b = new BottigliaConTappo(10);
        // upcast: da BottigliaConTappo a Bottiglia
        else b = new Bottiglia(10);
        // se oggi_piove=true allora b ha tipo vero BottigliaConTappo
        // Se oggi_piove=false allora b non ha tipo vero
        // BottigliaConTappo, ma Bottiglia

        // DOWNCASET: indicazione al compilatore che un certo oggetto
        // va considerato di un tipo più piccolo.
        // Non crea errori a runtime SOLO nel caso in cui
        // l'oggetto ha tipo vero che coincide con tale
        // tipo più piccolo.

        /*Il prossimo downcast appare corretto al compilatore, il
        quale non ha modo di sapere se il tipo vero 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 vero Bottiglia: */
    }
}
```

```

BottigliaConTappo t = (BottigliaConTappo) b;
// SE b ha tipo vero BottigliaConTappo, allora il downcast ha
successo anche se il suo tipo apparente è Bottiglia.
System.out.println( "Downcast avvenuto con successo");
// ALTRIMENTI il programma termina con una ClassCastException

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

```

### Esempio 2

```

//Bottiglia.java. Riutilizziamo il programma della Lezione 06

//BottigliaConTappo.java. Riutilizziamo il programma della
Lezione 12

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 istanziati 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
    // anche se null ha tipo sia BottigliaConTappo che
    Bottiglia perché ha tutti i tipi classe
}
}

```

### Esempio 3

```

//Bottiglia.java. Riutilizziamo il programma della Lezione 06

//BottigliaConTappo.java. Riutilizziamo il programma della
Lezione 12

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 per il compilatore, b ha tipo apparente
        // 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;
        }
    }
}

```

```

// compila e...
// ... 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() grazie al binding dinamico, siccome
// il tipo vero di b e z è BottigliaConTappo
// (certificato dalla instanceof)
}//fine if
}

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?
    */
}
}

```

Riassumendo, abbiamo visto che un downcast ha successo quando fa passare da un tipo apparente D a un tipo C più specifico di D:

- **a tempo di compilazione:** dati D tipo apparente di obj e C tipo del cast ((C) obj), si vuole che C <: D,
- **a runtime:** un downcast (C) obj con D tipo apparente di obj non dà errore a runtime solo se: *(i)* obj ha tipo vero C; *(ii)* oppure obj ha tipo vero C' <: C (sappiamo già che C

$\langle : D$ , se si è superata la compilazione, quindi per transitività  $C' \langle : D$ ).

Più avanti (precisamente dopo aver introdotto il costrutto di *interfaccia*), vedremo anche esempi di downcast corretti **con il tipo C non incluso in D**, ma con un sottotipo E in comune a C e D.

#### **Lezione14. Parte 2. Un esempio di ereditarietà e di binding dinamico.**

Mostriamo un semplice esempio in cui si può osservare l'effetto del binding dinamico durante l'esecuzione di un'applicazione. Si tratta di una applicazione che genera oggetti corrispondenti a due tipi di figure geometriche (i quadrati e i cerchi) e simula la loro rappresentazione con delle semplici stampe a video.

Le figure geometriche sono rappresentate dalla superclasse Figura, che astrae il comportamento rappresentando una figura "vuota", e dalle due sottoclassi Quadrato e Cerchio che specializzano Figura specificando come si disegnano i cerchi e i quadrati, rispettivamente.

Notate che si potrebbe fare l'esempio utilizzando una delle librerie grafiche di Java, tuttavia ciò sarebbe tecnicamente complesso da spiegare e non è parte di questo insegnamento. Pertanto, come detto, simuliamo il fatto che oggetti diversi disegnino forme diverse con delle "dichiarazioni di intenti" stampate a video, per esempio, "Simulo il disegno di un quadrato 4X4".

Segue il codice.

```
public class Figura {  
    // Dichiariamo il metodo di disegno draw() ma lo definiamo  
    // con body vuoto: serve solo per ricordarci di definire un  
    // metodo draw in ogni sottoclasse della classe Figura.  
    public void draw(){}
}  
  
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: ridefiniamo il metodo draw() di Figura per
disegnare un quadrato.
public void draw(){
    System.out.println("Simulo il disegno di un quadrato " +
lato + "X" + lato);
}

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

    // OVERRIDE: ridefiniamo il metodo draw() di Figura per
disegnare un cerchio.
    public void draw() {
        System.out.println("Simulo il disegno di un cerchio di
raggio " + raggio);
    }
}

public class TestFigure {
    public static void main(String[] args){
        int m=7,n=9;
        Figura[] figure = new Figura[n];
        // Array 'figure' di n oggetti (=null) di tipo Figura
        // Assegnamo le n figure: scegliamo m quadrati e (n-m)
cerchi
        // Possiamo farlo perche' il tipo di quadrati e cerchi è
sottotipo di Figura (upcast)
        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);
        // nei for facciamo degli upcast
        // "Disegnamo" le figure: sarà il tipo vero di figure[i]
che determinerà quale versione del metodo verrà chiamata
        for(int i=0;i<figure.length;++i)
            figure[i].draw();
    }
}

```

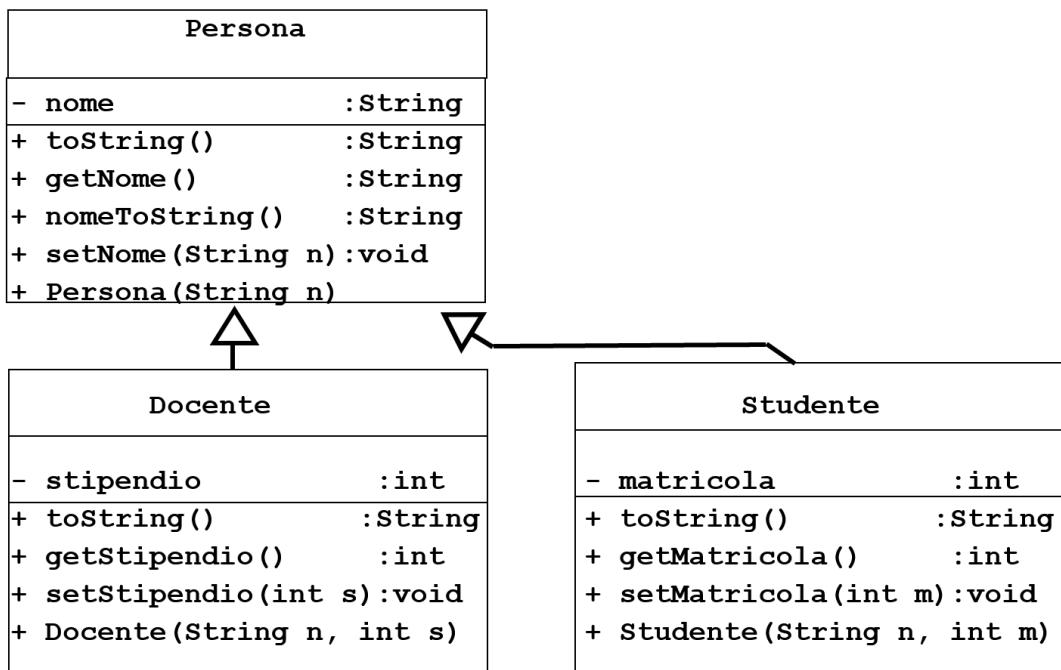
## Lezione 15

### Esempi di ereditarietà. Array come liste

**Lezione 15. Parte 1. Un altro esempio di uso del dynamic binding.** Nella lezione precedente, abbiamo introdotto una classe Figura con sottoclassi Quadrato e Cerchio, ciascuna contenente una versione specializzata del metodo draw() della superclasse, in modo da offrire modalità di disegno diverse a seconda del tipo vero della figura: il metodo da applicare veniva deciso con il meccanismo del dynamic binding. Vediamo ora l'esempio di una classe **Persona** con due metodi di conversione in stringa con body identici, **String toString()** e **String nomeToString()**. Riscriviamo, ovvero specializziamo, **toString()** in ciascuna sottoclasse di Persona, **Docente** e **Studente**, ma non riscriviamo **nomeToString()**. Proveremo quindi ad applicare i due metodi a un array di oggetti di tipo Persona. Nel caso di **toString()** il metodo invocato dipenderà dal tipo vero dell'oggetto di tipo apparente Persona. Nel caso di **nomeToString()**, invece, verrà sempre invocato il metodo della superclasse Persona, ereditato da entrambe le sottoclassi.

Segue un diagramma UML di Persona e delle sue sottoclassi Docente e Studente. Nella classe Persona abbiamo un attributo nome e i suoi metodi getter e setter, nelle classi Docente e Studente abbiamo un attributo in più, rispettivamente stipendio e matricola, con i rispettivi getter e setter.

### 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, specializzato in ciascuna sottoclasse
    public String toString(){
        return " nome = " + nome;
    }

    //Metodo di concatenazione di cui NON intendo fare OVERRIDE:
    //ereditato così com'è dalle sottoclassi
  
```

```

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

//Docente.java
public class Docente extends 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()
    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{
    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()
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 oggetti di tipo vero Docente o Studente, grazie al dynamic binding la JVM usa il metodo definito nel tipo più vicino (nel diagramma UML) al tipo vero. Nel caso di ***toString()*** il metodo usato è quello della sottoclasse Docente o Studente, nel caso di ***nomeToString()*** è quello della classe Persona.

```

//PersonaDemo.java
public class PersonaDemo {
    public static void main(String[] args){
        Studente a = new Studente("Rossi",111); //111=matricola
        Docente b = new Docente("Ferrero",1000); //1000= stipendio
        Persona c = new Persona("Bianchi");
        //Costruisco un array v contenente gli oggetti creati
        int n=3;
        Persona[] v = new Persona[n];
        v[0]=a; //upcast
        v[1]=b; //upcast
        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): viene
        chiamato il metodo più specializzato per ogni elemento a
        seconda che l'elemento abbia tipo vero Persona o Studente o
        Docente
        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):
viene chiamato il metodo più specializzato per ogni elemento,
che in questo caso è sempre quello della classe Persona
System.out.println( "\nEsempio di nomeToString()" );
for(int i=0;i<n;i++) {
    System.out.println(i + " " + v[i].nomeToString());
}
}
}
}

```

### Lezione 15. Parte 2. Le classi MiniLinkedList e Iterator.

Lasciamo momentaneamente da parte ereditarietà e dynamic binding per analizzare un caso di studio sulle **liste dinamiche**. Lo scopo di questo caso di studio è molteplice:

1. capire come inserire e cancellare un nodo in una posizione diversa dalla testa di una lista (che abbiamo visto per la classe DynamicStack nella Lezione 09);
2. introdurre una struttura-dati dinamica che abbia una forma di accesso via indice, come per gli array;
3. capire come evitare che la complessità della traversata della lista, nel caso di un'operazione applicata a tutti i nodi, diventi quadratica nel numero degli elementi.

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 totale dei nodi. La lista viene identificata con l'indirizzo **first** che è quello di  $v_0$ . Consideriamo indici logici (non *fisici*, come invece quelli degli array, per i quali rappresentano un offset a partire dall'indirizzo del primo elemento) le posizioni dei nodi nella lista, a partire dal primo nodo che per convenzione è quello in posizione 0. I metodi pubblici di MiniLinkedList sono:

1. **int get(int i)** Restituisce il contenuto del nodo in posizione i se  $0 \leq i < size$ .
2. **void set(int i, int x)** Assegna il valore x al nodo in posizione i, se  $0 \leq i < size$ .
3. **void add(int i, int x)** Aggiunge un nodo con elem x in posizione i, se  $0 \leq i \leq size$ , e incrementa size.
4. **int remove(int i)** Cancella il nodo di posizione 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 in posizione i. Questo metodo viene usato dai metodi pubblici. Tuttavia, non è pubblico perché, se lo fosse, consentirebbe il memory leak degli indirizzi dei nodi.

La struttura-dati MiniLinkedList assomiglia alla struttura-dati array, con il vantaggio che la dimensione size di un oggetto MiniLinkedList v non è fissata a priori, e che v occupa esattamente lo spazio di memoria di cui ha bisogno. C'è però uno svantaggio: l'operazione v.get(i) raggiunge il nodo i passando attraverso i nodi 0, ..., i-1. Pertanto, per restituire il valore del nodo in posizione i, usa un numero di getNext() uguale ad i. L'accesso a un array, invece, avviene con un solo accesso a un indirizzo.

Supponiamo ora che il nostro obiettivo sia di attraversare ciascun nodo della lista, **svolgendo per ognuno di essi la stessa operazione**. Per esempio, data una lista di 'size' elementi detta 'myList', si vuole scrivere un codice client per visualizzare a video i valori contenuti nei nodi della lista. Se la lista non offrisse un suo metodo pubblico print() che stampa a video tali dati, il codice client dovrebbe:

1. invocare il metodo size() (o getSize(), o analogo) della lista per conoscere size, ovvero quanti elementi ha la lista, e poi:
2. percorrere la lista per ottenere il valore di ciascun elemento di indice k, con  $0 \leq k < \text{size}$ , invocando sulla lista il metodo get(k), e stampare tale valore. Il codice risultante segue:

```
int lunghezza = myList.size();
for (int k=0; k<myList.size(); k++) {
    System.out.println( " " + myList.get(k));
}
```

Assumendo che il metodo get(i) della lista sia implementato in modo che parta dal nodo puntato da first della lista e la navighi fino alla posizione i, otterremo che:

1. get(0) è immediato: la lista restituisce il valore elem del suo first
2. get(1) richiede che, partendo dal first, la lista passi al secondo nodo invocando getNext() sul nodo first: 1 invocazione di getNext();

```
3. get(2): da first, getNext() per arrivare al nodo di posizione 1 e getNext() per arrivare al nodo di posizione 2: 2 invocazioni di getNext();  
4. e così via fino a k.
```

Questo compito richiede di invocare **0+1+2+...+(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. Si dice che la complessità di questo attraversamento della lista è **quadratico nel numero degli elementi, ovvero proporzionale a n^2**, con n il numero dei nodi. La sfida è di trovare un modo che permetta l'attraversamento della lista che sia proporzionale a n (cioè, nel nostro caso, a size).

**Si noti che aggiungere un metodo pubblico print() alla classe MiniLinkedList non sarebbe una buona scelta dal punto di vista dell'ingegneria del software.** Infatti, un tale metodo legherebbe la struttura-dati a una particolare visualizzazione degli suoi elementi, che non è detto sia quella voluta da chi usa la struttura-dati. Per ragioni come questa, l'output (e anche l'input) dei dati è buona norma che sia demandato al codice client o a librerie specifiche scelte dal programmatore del client stesso.

In alternativa, un modo semplice per permettere di eseguire operazioni su tutti i nodi della lista con complessità lineare, sarebbe rendere **pubblico** l'attributo first della lista stessa.

```
public class MyList {  
    public Node first; private int size;  
    // ...  
}
```

In tal modo, il codice client potrebbe eseguire un ciclo che, partendo da una puntatore ausiliario copia di first, venga aggiornato con l'indirizzo del nodo successivo fino a esaurimento dei nodi:

```
Node temp = myList.first;  
while (temp != null){  
    System.out.println(temp.getElem() + " ");  
    temp = temp.getNext();  
}
```

Questa tecnica però espone non solo il first, ma anche tutti i next dei nodi all'accesso da parte di codice esterno a un

oggetto di tipo MinLinkedList. Questi memory leak comportano il **rischio** che il programmatore di un codice client possa scrivere del codice che modifica la lista dall'esterno (ovvero senza utilizzare i metodi offerti da MiniLinkedList). Per esempio:

```
Node temp = myList.first;
while (temp != null){
    System.out.println(temp.getElem() + " ");
    temp.setElem(0); // azzerà il valore del nodo
    temp = temp.getNext();
}
```

modifica dall'esterno tutti i valori dei nodi della lista azzerandoli (mantenendo la stessa lunghezza della lista).

Modificare la lista dall'esterno potrebbe anche vanificare l'invariante di classe "size rappresenta il numero di elementi presenti nella struttura", se, per esempio, il programmatore si dimenticasse di modificare l'attributo size della lista quando aggiunge/toglie nodi direttamente, senza usare i metodi add() e remove() offerti da MyList.

Per consentire una scansione veloce della lista senza esporre i puntatori ai nodi introduciamo una classe **MinIterator**, i cui oggetti hanno una struttura analoga a quella degli oggetti di MiniLinkedList, ovvero hanno un campo privato, detto **next**, che è di tipo Node e che viene inizializzato dal costruttore con un valore Node.

```
public class MinIterator {
    private Node next;

    public MinIterator(Node n) {
        next = n;
    }
    // ...
}
```

La classe Iterator offre anche due metodi pubblici:

- **hasNext()**, per testare se il campo next è null (valore che indica l'aver raggiunto la fine di una struttura dinamica);
- **next()**, che fa avanzare il puntatore next al nodo successivo, se hasNext() restituisce il valore true.

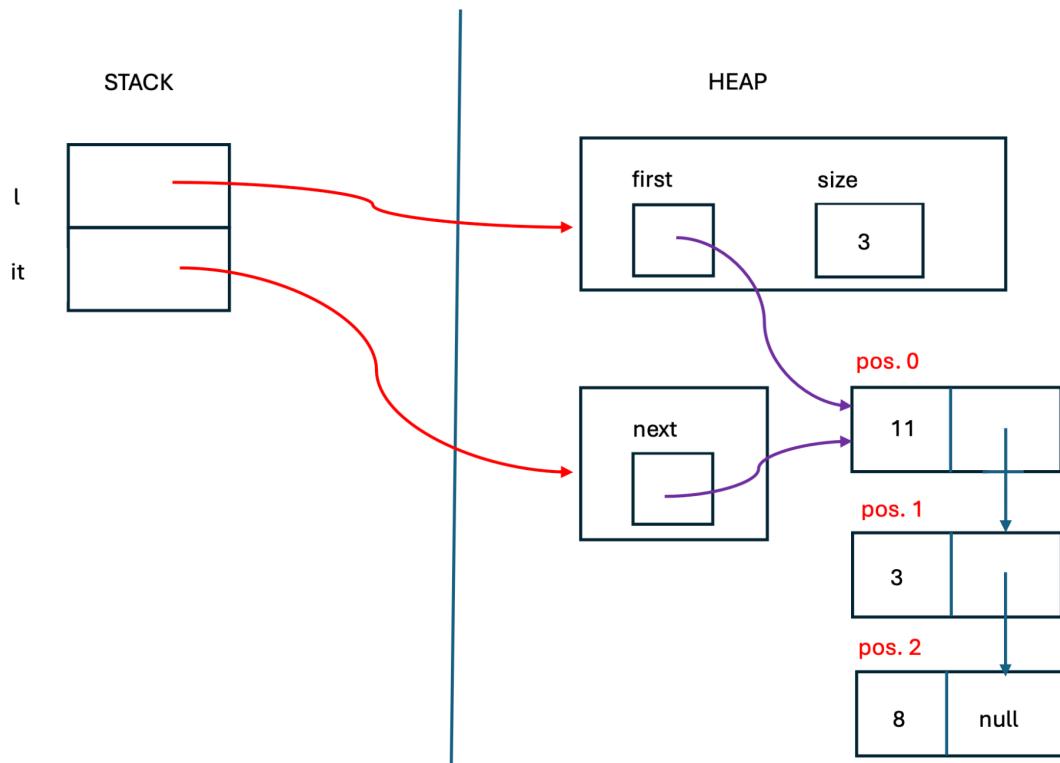
Il collegamento tra le due classi MiniLinkedList e MiniIterator è dato da un metodo **iterator()** appartenente alla classe MiniLinkedList. Questo metodo crea un oggetto di tipo MiniIterator, inizializzato con il valore del first dell'oggetto di tipo MiniLinkedList.

```
public class MiniLinkedList {
    private Node first;
    private int size;

    // ... metodi vari di MiniLinkedList

    public MiniIterator iterator(){
        return new MiniIterator(first);
    }
}
```

Il disegno che segue illustra questo collegamento.



Il MiniIterator è un puntatore agli elementi della lista MiniLinkedList utilizzabile da codice client esterno a MiniLinkedList. Scorrendo la lista attraverso il MiniIterator, il codice client accede ai suoi nodi senza perdere il first della lista, che può continuare a puntare al primo nodo.

**Facciamo ora una piccola parentesi per confrontarci con DynamicStack (Lezione 09) e con MyList riportata sopra.**

**Per quanto riguarda DynamicStack**, il MiniIterator ha la stessa funzione della variabile "temp" che abbiamo utilizzato nel metodo `toString()` della classe DynamicStack per non distruggere la pila modificando il suo `first` - riprendiamo qui un estratto del codice di DynamicStack:

```
public class DynamicStack{
    // ...
    public String toString(){
        Node temp = top; //partiamo dal nodo in cima alla pila
        String s = "";
        while (temp != null){
            s=s+" "+temp.getElem()+"\n";
            temp=temp.getNext();
        }
        return s;
    }
}
```

Tuttavia, DynamicStack e MiniIterator differiscono profondamente sulla modalità di memorizzazione della variabile che punta ai nodi delle liste:

- In DynamicStack, `temp` è una variabile dichiarata all'interno del metodo `toString()` e viene allocata nello stack della JVM durante l'esecuzione del metodo. Solo `toString()` può usare tale variabile per fare il suo lavoro.
- Invece, per quanto riguarda MiniIterator, lo stack della JVM contiene il riferimento all'oggetto MiniIterator (non al suo `next!`), utilizzabile da codice client. Il `next` di MiniIterator è una variabile privata che viene memorizzata nella heap della JVM, all'interno dell'area dedicata all'oggetto.

Per quanto riguarda in confronto con MyList, che esponeva il `first` della lista come variabile pubblica, MiniIterator nasconde "next" come variabile di istanza privata. Pertanto:

- In MyList, il codice client può accedere al `first` della lista e copiarlo in una sua variabile di programma (che sta sullo stack) per manipolarlo.

- In MiniIterator, non si permette al codice client di copiare il contenuto della variabile next in una variabile di programma (che sta sullo stack), cioè permette di evitare il leak di first e degli altri nodi della lista.

**Fine della parentesi!**

**Tornando a MiniLinkedList:** la motivazione principale per cui abbiamo introdotto il **MiniIterator** è che **migliora l'efficienza nella gestione delle operazioni che richiedono di scorrere le liste** (come la stampa di tutti gli elementi di una lista): ci permette infatti di attraversare una lista e leggere il contenuto dei suoi nodi in un numero di passi lineare rispetto al numero dei nodi stessi.

Mostriamo ora il codice di MiniLinkedList e MiniIterator.

```
// Node.java Riutilizziamo la classe Node della Lezione 09

// 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 negli altri metodi della classe. Non
    // viene reso pubblico per evitare che dall'esterno sia
    // possibile modificare i nodi della classe senza aggiornare size
    // (-> per evitare memory leak). */
}

private Node node(int i){
```

```

//controllo che i sia un posizione "legale" della lista
assert 0 <= i && i < size: "i non in [0,size-1]";
//creo una copia di first per non modificarlo
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 questa 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){
        //aggiungo un nodo all'inizio (come per DynamicStack)
        first = new Node(x, first);
    }
    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()));
    // OPPURE
    // Node vecchio_node_i = prev.getNext()
    // Node nuovo_node_i = new Node(x, vecchio_node_i);
    // prev.setNext(nuovo_node_i);
}
//l'invariante di classe e` temporaneamente non valido:
size vale uno meno il numero di elementi della lista. Quindi
incrementiamo di 1 size:
size++;
}

/** remove(i) elimina il nodo in posizione i e ne restituisce
il contenuto */
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 = nodo precedente al 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.getElem();
        //per eliminare il nodo i, puntato da 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 a size:
    size--;
    return x;
}

```

```

/* Definiamo un metodo che copia il first di una
MiniLinkedList nel campo next di un oggetto di classe
MiniIterator */
public MiniIterator iterator(){
    return new MiniIterator(first);
}
// Per visitare una MiniLinkedList l scriverò MiniIterator
it = l.iterator() e farò muovere it usando i metodi hasNext()
e setNext() della classe MiniIterator, si veda sotto nella
classe TestMiniIterator.
}

// MiniIterator.java
public class MiniIterator {
    private Node next;

    public MiniIterator(Node n){
        next = n;
    }

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

    /* Metodo next() che restituisce l'elemento nel nodo corrente
    e muove il campo next al nodo dopo. Si noti che next()
    cancella il valore precedente 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(){

        int x = next.getElem();
        next = next.getNext();
        return x;
    }
}

// TestMiniIterator.java
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 sopra, perché ogni get(i), per ogni i, ricomincia dal
nodo in posizione 0 (quello puntato da first).
    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 un "memory leak".
*/
}

//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());
    }

}

```

# Lezione 16

## Classi astratte di figure

Una classe **Figura** per il calcolo di area e perimetro delle **figure geometriche**. In questa lezione definiamo un insieme di classi per **calcolare area e perimetro** delle figure geometriche: cerchi, poligoni regolari, trapezi, rettangoli. Tramite questo esempio introdurremo le **classi astratte** di Java.

Una prima soluzione usando solo la nozione di **sottoclasse**. Similmente a quanto fatto nella Lezione 14, possiamo definire una superclasse **Figura** che definisca dei metodi con body vuoto per calcolare area e perimetro, **area()** e **perimetro()**. L'utilità di questi metodi, pur vuoti, è garantire che ogni sottoclasse di Figura li erediti. In pratica, **area()** e **perimetro()** entrano così a far parte dell'interfaccia pubblica di **Figura** e delle sue sottoclassi. Le sue sottoclassi specifiche ridefiniranno poi tali metodi con body adatti a calcolare area e perimetro di ciascun tipo di figura. Segue una parte del codice di questa soluzione.

```
public class Figura {

    public double area() {return 0;}
    public double perimetro() {return 0;}

}

// end class Figura

/** Cerchio.java
Contiene la sottoclasse Cerchio di classe Figura:
- ha un attributo e metodi specifici per il raggio del
cerchio
- sovrascrive area() e perimetro(), */

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

```

Questa soluzione comporta uno svantaggio: **non c'è garanzia che tutte le sottoclassi di Figura implementino una versione specifica dei metodi area() e perimetro()** facendo l'override dei rispettivi metodi della superclasse, poiché non ci sono vincoli formali che il compilatore possa controllare. Potrebbe quindi capitare che, invocando area() o perimetro() su un oggetto che appartiene a una sottoclasse, venga eseguito il metodo ereditato, con body vuoto.

**Una soluzione che usa il costrutto classe astratta.** Possiamo migliorare la soluzione precedente dichiarando la classe Figura e i suoi metodi area() e perimetro() come metodi **astratti** (utilizzando la parola-chiave **abstract**). Una classe è astratta (e deve anch'essa essere dichiarata abstract) se ha almeno un metodo astratto. Una classe astratta non si può istanziare, ovvero non possiamo creare oggetti di tipo Figura, ma possiamo definire delle sue sottoclassi *istanziabili*, rappresentanti figure geometriche particolari, che sovrascrivono i metodi astratti ereditati implementandone il body. I metodi astratti rappresentano un vincolo che il programmatore delle sottoclassi deve necessariamente rispettare implementandoli. Chiameremo **concreta** una classe non astratta.

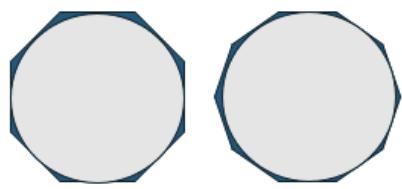
### **Regole per le classi astratte.**

1. I campi non sono astratti.
2. Se un **metodo è astratto la sua classe è astratta**, ma una classe astratta può avere anche metodi concreti (se serve avere metodi comuni a tutte le sottoclassi, che li erediteranno).
3. Una classe astratta può **definire dei costruttori**, ma questi si possono **invocare solo nei costruttori nelle sottoclassi**. I costruttori servono per inizializzare le variabili che sono definite all'interno della classe astratta (ricordate che le variabili private non sono accessibili alle sottoclassi).
4. Perché una sottoclasse di una classe astratta sia concreta deve **sovrascrivere tutti i metodi astratti** che eredita.

Forniamo ora il **diagramma UML della gerarchia che parte dalla classe Figura** di figure, per le quali vengono forniti i metodi di calcolo di area e perimetro. Questi sono i soli metodi (astratti) di Figura. Nei diagrammi UML, metodi astratti e classi astratte sono solitamente indicati scrivendone il nome in ***corsivo*** (noi inoltre li indicheremo in **rosso**).

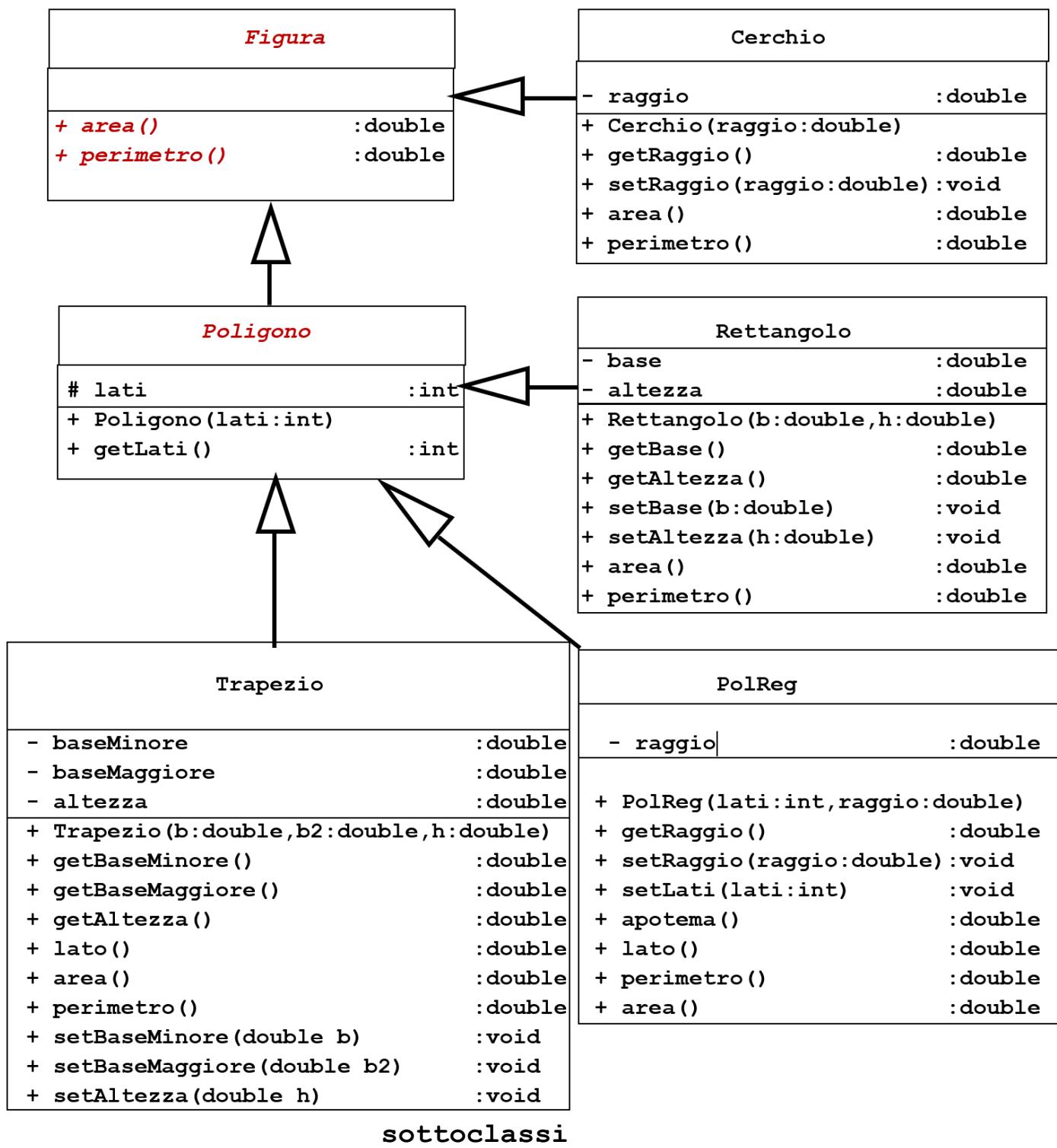
Le sottoclassi dirette di Figura sono Cerchio (classe concreta), avente l'attributo raggio e relativi get() e set(), e Poligono (classe astratta), con attributo lati (che rappresenta il numero dei lati) e un metodo get(). Non possiamo fornire un metodo set() per il numero dei lati del Poligono, perché ogni poligono ha un numero di lati specifico. Per questo motivo, Poligono è essa stessa una classe astratta, anche perché non abbiamo un metodo per il calcolo di area e perimetro che valga per ogni poligono.

Le sottoclassi di Poligono sono invece 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 inseriamo un metodo setLati(int lati) per variare il numero dei lati, assumendo che sia di utilità alle applicazioni. Per esempio, pensando ad una applicazione grafica, si potrebbe fare un gioco in cui, a parità di circonferenza inscritta nel poligono regolare, si cambia il numero di lati del poligono periodicamente, come nella figura seguente:



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



Segue il codice Java.

```
//Figura.java

/** LA CLASSE ASTRATTA FIGURA */
public abstract class 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();

    public static void main(String[] args){
        // Posso dichiarare variabili di tipo Figura:
        Figura p; //OK
        /** Però NON posso creare oggetti di tipo Figura con new
        Figura(). Invece, potrei assegnare a p un oggetto di una
        sottoclasse concreta di Figura, oppure null.
        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 concreta di Figura:
sovrascrive area() e perimetro(), ha un attributo e metodi
specifici per il raggio del cerchio */

public class Cerchio extends Figura {
    private double raggio; //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;
    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, pur avendo degli
attributi e un metodo di lettura per il numero dei lati. */

public abstract class Poligono extends Figura {
    protected int lati; // INVARIANTE: lati>=3
    /* "protected" consente di modificare "lati" nelle
    sottoclassi di Poligono (in particolare, ci serve per i
    poligoni regolari) */

    public Poligono(int lati){
        //Non e' necessario invocare il costruttore della classe
        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 in questa classe niente metodo
set()

    public static void main(String[] args){
        // Posso dichiarare variabili di tipo Poligono:
}

```

```

Poligono p;
//Poligono t = new Poligono(3);
//NON corretto, Poligono è una classe astratta
/** NON posso creare oggetti di tipo Poligono con new
Poligono(), ma potrei assegnare a p un oggetto di una
sottoclasse concreta di Poligono, oppure null. ATTENZIONE: il
valore di default di p non è null. Infatti se si prova a
stampsare 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 concreti per calcolare area e perimetro, piu'
attributi e metodi specifici per raggio, lato e apotema.
Promemoria: l'apotema e' il raggio della circonferenza
inscritta e serve per calcolare l'area del poligono regolare.
*/
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;
    }

    public double getRaggio(){
        return this.raggio;
    }

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

    //Il numero dei lati di un poligono regolare puo' cambiare.
    public void setLati(int lati){ //richiede "lati" protected
nella classe Poligono
        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);
}

/** Questo main serve a far vedere che posso assegnare un
oggetto e della sottoclasse concreta PolReg a una variabile di
tipo Figura (classe astratta) perche' questa sottoclasse non
e' astratta. Il tipo esatto di E determina il metodo usato per
calcolare l'area di E. */
public static void main(String[] args){
    Figura e = new PolReg(6,1.0);
    System.out.println(e.area());
    //calcolata con la versione del metodo della classe PolReg
}
}

//Trapezio.java

/** Sottoclasse concreta 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 */
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 concreta di Poligono e quindi di Figura.
Rettangolo eredita getLati() dalla classe Poligono. 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
/** 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. */

```

```

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());
        /** Posso definire un array con figure di OGNI TIPO */
        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 17

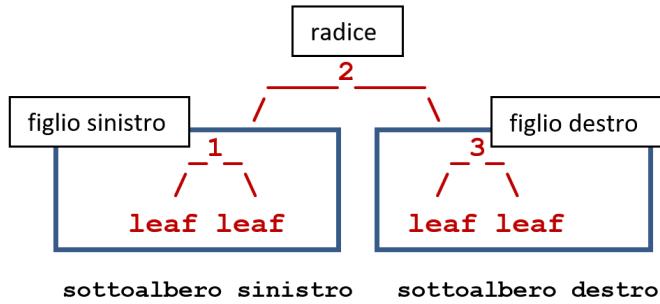
### La classe astratta ricorsiva degli alberi di ricerca

**Definizione di albero binario.** Vediamo ora un esempio di classe astratta **e ricorsiva**: la classe degli **alberi di ricerca**. Si tratta di una classe importante per definire una struttura dati gerarchica 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. In particolare, vedremo alberi di ricerca di interi, che rappresentano insiemi finiti di interi, ma con poco più lavoro definiremo più avanti una classe generica di alberi di ricerca i cui valori hanno tipo T parametrico. E' infatti utile poter utilizzare alberi di ricerca per memorizzare dati di qualunque tipo in modo da potervi accedere in modo efficiente.

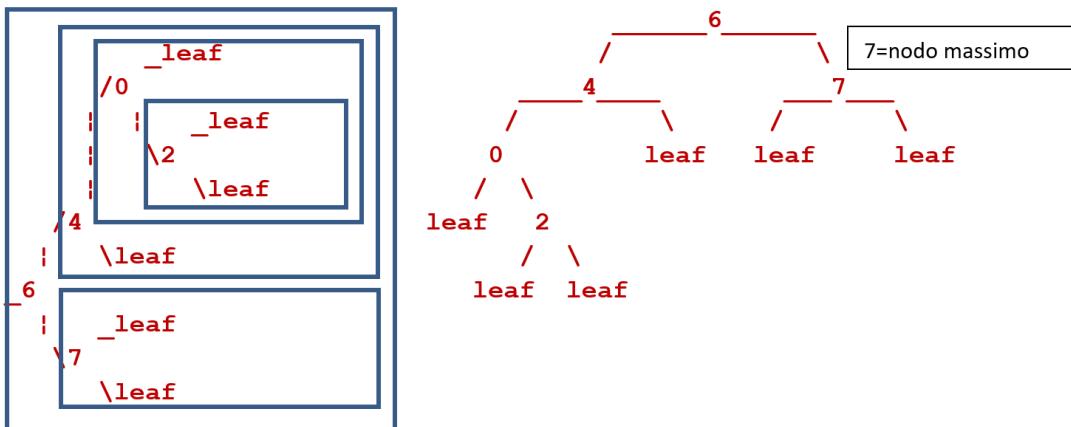
Iniziamo con la definizione ricorsiva degli **alberi binari di interi**. Gli alberi binari di interi sono **alberi vuoti**, rappresentati **con oggetti di tipo Leaf**, oppure **hanno una radice**, costituita da un intero, e un **sottoalbero sinistro** e uno **destro**. Le radici dei sottoalberi sono 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.

**IMPORTANTE!** In questa versione di albero binario di ricerca **l'albero vuoto non è rappresentato da null, bensì da un oggetto Leaf**. Questa rappresentazione permette un utilizzo del binding dinamico nei metodi ricorsivi molto naturale.

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. Non spieghiamo la complessità della ricerca negli alberi binari di ricerca perché viene approfondita nell'insegnamento Algoritmi e Strutture Dati. Come esempio, ecco un albero binario di ricerca 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 segue queste regole: la classe astratta e ricorsiva Tree ha due sottoclassi concrete, **Leaf** (alberi vuoti) e **Branch** (alberi non vuoti), gli elementi di Branch possono contenere come sottoalberi sinistro e destro elementi di tipo Leaf e altri elementi di tipo Branch.

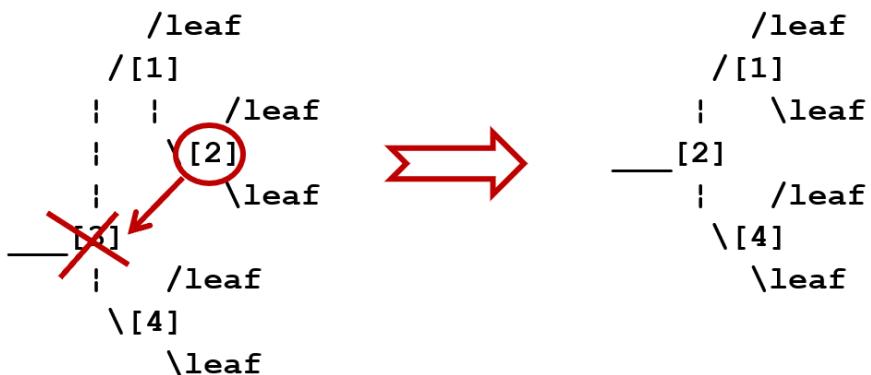
Gli elementi di Tree hanno la caratteristica di essere **dinamici**, cioè cambiano di dimensione a seconda della

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. Nell'insegnamento PPOO non ci occupiamo di questo secondo aspetto, che sarà trattato in Algoritmi e Strutture Dati.

Come metodi della classe Tree definiamo:

1. **boolean empty()** per sapere se un albero è vuoto,
2. **int max()** per calcolare il nodo massimo di un albero,
3. **boolean contains(int x)** per sapere se un albero contiene il nodo **x**,
4. **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.
5. Infine aggiungiamo un metodo **String toString()** per convertire l'albero in una String che permette di "disegnare" un albero con soli caratteri ASCII. Non spiegheremo questo metodo ma lo includiamo perché permette delle visualizzazioni a video leggibili degli alberi.

**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 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 che 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 (quello della precedente radice). 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. Si noti anche che il contenuto precedente di un albero **NON è ricostruibile**.

```
// Tree.java - Classe astratta degli 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, istanza di Leaf, che
// chiamiamo per brevità "leaf"
// BRANCH: contiene tutti gli alberi non vuoti

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);
// Nel codice client va usato 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);
// Nel codice client va usato 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.16

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. */
}

```

---

<sup>16</sup> 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()`.

```

// end class Tree

//Leaf.java - I suoi oggetti rappresentano alberi vuoti

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

public class Leaf extends Tree {
    public Leaf() { }
    // Si noti che nella classe Leaf non ci sono attributi, poiché un nodo che rappresenta un albero vuoto non contiene nessun valore e non ha nessun sottoalbero. Il costruttore di Leaf perciò non inizializza nulla e si potrebbe anche lasciare implicito. Un oggetto di questa classe è quindi soltanto un segna-posto, che serve a indicare i nodi terminali di un albero, ovvero un albero vuoto.
    // Ribadiamo che in questa rappresentazione NON si usa null per indicare un albero vuoto, bensì proprio un oggetto di classe Leaf, in modo da sfruttare il più possibile il binding dinamico nei metodi. Un albero viene inizializzato vuoto con new Leaf() e poi esteso un elemento alla volta. Qui this = oggetto istanziato = albero vuoto (che non e' null). */
}

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.
Tuttavia 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; in questa classe this è un oggetto di tipo Leaf: lo riuso, quindi non creo alberi vuoti nuovi.

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
conversione a String. Non forniamo spiegazioni sul suo
funzionamento. Notiamo solo che, come rappresentazione in
stringa di caratteri, utilizza il nome "leaf" che abbiamo
scelto di dare all'albero vuoto.

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

// end class Leaf

/** Branch.java - 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(); }
} // if (right.empty()) return elem; else return 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 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
conversione in String. 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 visualizzare a video un albero t, trascrivo t nella
stringa t.toString(), con toString() ridefinito. Il comando
e': System.out.println(t), solita 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(20);

    //r e` un generatore di numeri pseudo-casuali.

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

// end class Test Tree

```

```

/* 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

**Visita di un albero binario in pre-, in-, post-ordine e per livelli. Note sul metodo equals()**

**Lezione 18. Parte 1. Visite degli alberi binari.** In questa lezione presentiamo alcuni metodi notevoli per considerare in modo esaustivo tutti i nodi di un albero, le **visite**. Ci sono diversi ordini di visita: *pre-ordine*, *in-ordine*, *post-ordine* e *per livelli*. Imparare a visitare un albero in modo esaustivo è fondamentale per permettere di effettuare operazioni su tutti i suoi nodi senza dimenticarne nessuno. Per esempio, li si potrebbe dover contare, modificare secondo il loro valore secondo un certo criterio, visualizzare a video i valori dei nodi, e così via. Proporremo le visite per calcolare il numero di nodi di tipo Leaf in ogni **livello** dell'albero. Un **livello** è fatto di tutti i nodi alla stessa distanza dalla radice dell'albero.

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, ed è particolarmente congeniale effettuarle con metodi ricorsivi. La **visita per livelli** è invece quella in cui 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. Quest'ultima visita è più facile da realizzare in modo iterativo.

Vediamo ora il codice Java delle visite e del calcolo del numero dei nodi di un albero.

```
//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". Durante la
visita, questi metodi visualizzano a video i valori contenuti
nei nodi.
 */
//visita e visualizza a video i nodi in pre-ordine:
//radice-sinistra-destra
public abstract void preOrder();

//visita e visualizza a video i nodi in in-ordine:
//sinistra-radice-destra
public abstract void inOrder();

//visita e visualizza a video i nodi in post-ordine:
//sinistra-destra-radice
public abstract void postOrder();

//restituisce sotto forma di String i valori di tutti i
nodi a un dato livello n, scorrendoli da sinistra a destra
public abstract String livello(int n);

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

//calcola il numero di oggetti di tipo Leaf
//di un albero che sono 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 conversione di un
oggetto Tree a String.

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

// metodo pubblico di conversione Tree -> String,
dall'alto verso il basso, con i sottoalberi situati piu' a
destra dell'albero di cui fan parte. Predisponde un disegno
bidimensionale dell'albero, 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;}

    public void inOrder() {}

    public void preOrder() {}

    public void postOrder() {}

    /* Visita per livello */
    public void livello(){}

    /* Metodo ricorsivo ausiliario usato da livello() */
    public String livello(int n){return "";}
        //un oggetto Leaf non contiene elemento
        //per questo si restituisce la stringa vuota

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

    //Metodo che gestisce la parte NON pubblica della
    conversione a String.
    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; // figli sinistro e destro

    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 e
    visualizzo a video il suo valore, scendo a destra */

```

```

public void inOrder() {
    left.inOrder();
    System.out.print(elem + " ");
    right.inOrder();
}

/* Visita in pre-order: visito la radice e visualizzo a
video il suo valore, 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 e visualizzo a video il suo valore */
public void postOrder() {
    left.postOrder();
    right.postOrder();
    System.out.print(elem + " ");
}

/* Visita per livello */
public void livello() {
    int liv=0;
    String s = livello(liv);
    // ora s contiene i valori dei nodi a livello 0
    while (s.length() > 0){
        System.out.println(s);
        liv++; // scendo di un livello
        s = livello(liv);
        // ora s contiene i valori dei nodi del livello liv
    }
}

/* Metodo ricorsivo ausiliario usato da livello().
Restituisce sotto forma di String i valori di tutti i nodi a
un dato livello n, qui indicato con il parametro path,
scorrendoli da sinistra a destra. Abbiamo scelto di chiamare
il parametro path, anziche' n, perche' per raggiungere il
livello desiderato si conta il numero di archi da navigare per
raggiungere il livello richiesto. In particolare:
- la radice sta a livello 0 e per raggiungerla si percorre un
path di lunghezza 0;

```

```

- i nodi a livello 1 richiedono di scendere sul figlio
sinistro della radice e sul destro. In entrambi i casi, al
momento della chiamata ricorsiva, il path da percorrere ha
lunghezza 0.
- e cosi' via.
*/
public String livello(int path) {
    if (path==0) return elem + " ";
    else return left.livello(path-1) + right.livello(path-1);
}

public int leavesAt(int path){
    if (path==0)
        return 0;
    else
        return left.leavesAt(path-1) + right.leavesAt(path-1);
}

//Metodo che gestisce la parte NON pubblica della conversione
Tree -> String.
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.java
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 Leaf())));
}
}

```

```

        new Branch(9,
                    new
Leaf(), 
                    new
Leaf()))));

```

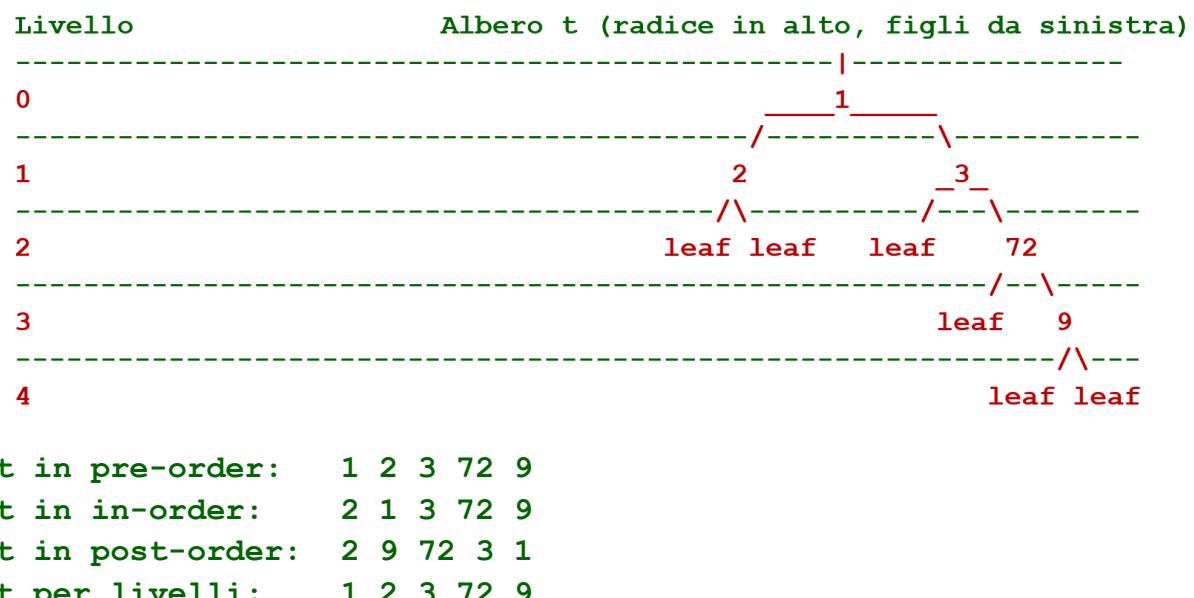
*/\* Albero t (radice a sinistra, figli dall'alto in basso). Radice 1, figlio sinistro 2, figlio destro 3, che ha figlio destro 72, che ha figlio destro 9.*

```

        /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.



```

*/
```

```

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: leaf per livello = 0 0 3 1 2 0 */
}
}

```

Le visite sono utilizzate in diversi algoritmi che coinvolgono gli alberi. Per esempio:

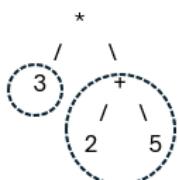
- Per disegnare un albero da sinistra a destra facciamo comparire i nodi in in-ordine. Si noti che **una visita in-ordine di un albero binario di ricerca visita l'albero seguendo la relazione d'ordine codificata nell'albero.** Per esempio, se visitiamo in-ordine l'albero seguente, l'ordine di accesso ai nodi dell'albero è il seguente (mostriamo il contenuto dei nodi per identificarli): 8, 12, 28, 35, 50, 56, 77, 80, 90, 92.

```

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

```

- Per valutare una espressione algebrica valutiamo in post-ordine l'albero delle sue sotto-espressioni: dobbiamo prima valutare le sottoespressioni e solo dopo combinarle con l'operatore che sta nella radice.



- 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.

Es: nel gioco degli scacchi, in un certo stato potrei muovere un pedone, la torre a destra o a sinistra, l'alfiere in una sola direzione. Quale scelgo? Devo esaminare tutte le possibilità che ho.

**Lezione 18. Parte 2. Note sul metodo `equals()`.** Abbiamo già scritto il metodo `equals()` per alcune classi nelle lezioni precedenti. È un **metodo binario**, ovvero un metodo che, *in linea di principio*, prende come parametro un oggetto `obj` della stessa classe dell'oggetto `this` su cui viene invocato il metodo. In particolare, il metodo `equals()` è un metodo binario

che restituisce un valore boolean: `true` se l'oggetto `this` è uguale all'oggetto `obj` passato come parametro, `false` altrimenti. Il criterio di uguaglianza è stabilito di volta in volta dal programmatore del metodo stesso. Per esempio, nella classe `Animal` della Lezione 04 avevamo la seguente implementazione del metodo `equals()`:

```
public boolean equals(Animal altroAnimale){  
    return  
        (this.nome.equalsIgnoreCase(altroAnimale.nome))  
        &&  
        (this.eta == altroAnimale.eta)  
        &&  
        (this.peso == altroAnimale.peso);  
}  
}
```

In questo caso il parametro di `equals()` è proprio di tipo `Animal`. Allora perché sopra abbiamo detto "*in linea di principio*"? Adesso che abbiamo visto l'ereditarietà e l'override, possiamo capire meglio la natura del metodo `equals()`.

Il metodo `equals()`, come il metodo `toString()`, fa parte dei metodi della classe `Object`, la superclasse di tutte le classi Java, comprese quelle che scriviamo noi, anche se non lo indichiamo mai esplicitamente con una "extends `Object`". La firma del metodo `equals()` nella classe `Object` è la seguente:

```
public boolean equals(Object obj)
```

e la politica di confronto stabilita dai progettisti Java è tale per cui esso restituisce `true` se l'indirizzo nella heap dell'oggetto `this` è uguale a quello di `obj`, ovvero se `obj` è un alias di `this`, `false` altrimenti.

La domanda che ci facciamo ora è: il metodo `equals()` della classe `Animale` che abbiamo riportato sopra è un **override** del metodo `equals()` della classe `Object`? **La risposta è no.** La ragione è che **le due firme sono diverse**, in particolare il parametro di `equals()` nella classe `Object` è di tipo `Object` e il parametro di `equals` nella classe `Animal` è di tipo `Animal` (ricordiamo che i nomi dei parametri sono *influenti*):

```
public boolean equals(Object obj)  
  
public boolean equals(Animal altroAnimale)
```

Se si volesse fare l'override del metodo `equals()` di `Object`, allora sarebbe giusto scrivere:

```
public boolean equals(Object altroAnimale) {  
    return  
        (this.nome.equalsIgnoreCase(altroAnimale.nome))  
        && (this.eta == altroAnimale.eta)  
        && (this.peso == altroAnimale.peso);  
}
```

?

No, perché il compilatore darebbe errore, visto che il tipo apparente di `obj` è `Object` e `Object` non ha gli attributi `nome`, `eta` e `peso`!

Ci viene però in aiuto il predicato `instanceof` che controlla a runtime il tipo vero di un oggetto:

```
public boolean equals(Object altroAnimale){  
    if (altroAnimale instanceof Animal){  
        // se altroAnimale ha tipo vero Animal posso fare il  
        confronto campo per campo, altrimenti se obj è null o ha altro  
        tipo restituisco false. Si noti che occorre mettere il  
        downcast (Animal) per dire al compilatore che in quel caso il  
        parametro 'altroAnimale' di tipo Object va invece considerato  
        di tipo Animal.  
        Animal a = (Animal)altroAnimale;  
        return (this.nome.equalsIgnoreCase(a.nome))  
            && (this.eta == a.eta)  
            && (this.peso == a.peso);  
    }  
    else  
        return false;  
}
```

Possiamo ora chiederci che relazione c'è tra il metodo `equals(Animal altroAnimale)` (che sta nella classe `Animal` della Lezione 04 e che abbiamo riportato sopra) e il metodo `equals(Object altroAnimale)`, di cui si hanno due versioni, una in `Object` e una che ne fa l'override in `Animal` (ricordiamo che i nomi dei parametri sono *influenti*).

I due metodi `equals()` sono in overloading, ovvero sono due metodi che hanno lo stesso nome ma che implementano operazioni

**potenzialmente diverse.**

Ricordiamo che se ci sono più versioni di un metodo in override viene applicato a runtime il binding dinamico, che sceglie la versione che corrisponde al tipo vero dell'oggetto. A differenza dell'override, **l'overloading non è risolto a runtime ma a tempo di compilazione**: il compilatore sceglie staticamente la versione da applicare in base al tipo dei parametri della chiamata.

Esempi di overloading che già vi sono noti sono l'operatore '+', che viene interpretato come somma aritmetica o come concatenazione (di stringhe) a seconda del tipo degli argomenti, o i costruttori di Java: quando in una classe sono definiti più costruttori, il compilatore sceglie quello da applicare in base al tipo e/o numero degli argomenti; se ci sono ambiguità, il compilatore dà errore.

Vediamo ora un codice per sperimentare con questi concetti.

```
//Animal.java
public class Animal{
    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 Animal(String n, int e, double p)
        {nome = n; eta = e; peso = p;}

    public String toString()
        {return " |nome=" + nome + "\n eta=" + eta + "\n peso=" +
peso;}}
```

```

/* Il metodo che segue fa l'override del metodo equals() di
Object: se l'oggetto su cui chiamiamo questo metodo ha tipo
vero (a runtime) Animal (o un suo sottotipo) verrà chiamata
questa versione, altrimenti verrà invocata la versione di
Object. */
public boolean equals(Object altroAnimale){
    // se altroAnimale ha tipo vero Animal posso fare il
    confronto campo per campo, altrimenti se obj è null o ha altro
    tipo restituisco false
    if (altroAnimale instanceof Animal){
        Animal a = (Animal)altroAnimale;
        return (this.nome.equalsIgnoreCase(a.nome))
            && (this.eta == a.eta)
            && (this.peso == a.peso);
    }
    else
        return false;
}

/* Il metodo seguente NON fa override del metodo equals() di
Object, bensì rende il metodo equals() un metodo overloaded.
*/
public boolean equals(Animal altroAnimale){
    return (this.nome.equalsIgnoreCase(altroAnimale.nome))
        &&
        (this.eta == altroAnimale.eta)
        &&
        (this.peso == altroAnimale.peso);
}

/* Il compilatore sceglie, a seconda del tipo dell'argomento
'altroAnimale', tra le due versioni di equals() in overload,
in base al tipo del parametro esplicito. Ci sono infatti due
versioni di equals() con due firme, una è quella con il
parametro formale 'altroAnimale' di tipo Animal e l'altra è
quella con il parametro formale 'altroAnimale' di tipo Object,
ereditata da Object e di cui la classe Animal fa overriding.
*/

// ProvaOver.java
public class ProvaOver{

    public static void main(String[] args){

```

```

// Quali metodi equals() vengono invocati? Che risultati
otteniamo? Perché? Provare a pensare a altri esempi.

Animal a = new Animal("n", 2, 3);
Animal b = new Animal("m", 3, 4);

// (1) invoca la versione overloaded equals(Animal
altroAnimale) in Animal (false):
System.out.println("Esp. 1 - " + a.equals(b));

Object c = new Animal("o", 5, 1);
Object d = new Animal("o", 5, 1);

// (2) invoca equals(Object altroAnimale) in Animal (true):
System.out.println("Esp. 2 - " + c.equals(d));

Object e = new Object();

// (3) invoca equals(Object altroAnimale) in Object
(false):
System.out.println("Esp. 3 - " + e.equals(d));

// (4) invoca equals(Object altroAnimale) in Animal
(false):
System.out.println("Esp. 4 - " + d.equals(e));

Object f = d;

// (5) invoca equals(Object altroAnimale) in Animal
(true):
System.out.println("Esp. 5 - " + f.equals(d));

// (6) invoca equals(Object altroAnimale) in Animal
(true):
System.out.println("Esp. 6 - " + ((Animal)f).equals(c));

// (7) invoca equals(Object altroAnimale) in Object
(true):
System.out.println("Esp. 7 - " + e.equals(e));

// (8) invoca equals(Object altroAnimale) in Animal
(false):
System.out.println("Esp. 8 - " + f.equals(e));

```

```

// (9) invoca equals(Object altroAnimale) in Object
(false):
System.out.println("Esp. 9 - " + e.equals(f));

// (10) invoca equals(Object altroAnimale) in Animal
(false, per via dell'instanceof):
System.out.println("Esp. 10 - " + a.equals(e));

// (11) invoca equals(Object altroAnimale) in Object
(false, perché gli indirizzi sono diversi):
System.out.println("Esp. 11 - " + e.equals(a));

// (12) invoca equals(Object altroAnimale) in Object
(false, perché gli indirizzi sono diversi):
System.out.println("Esp. 12 - " + e.equals((Object)a));
}
}

```

**Nota avanzata su overloading e overriding.** Prima abbiamo spiegato che le firme:

```

public boolean equals(Object obj)

public boolean equals(Animal altroAnimale)

```

sono diverse e quindi il metodo equals(Animal ...) non è un override del metodo equals(Object ...) ma è un overloading. Questa affermazione è corretta. Tuttavia, consideriamo le seguenti domande:

1. Ci sono casi in cui si potrebbe **fare override di un metodo cambiando il tipo del parametro della nuova versione del metodo nella sottoclasse senza introdurre errori?**
2. Ci sono casi in cui si potrebbe **fare override di un metodo cambiando il tipo di ritorno della nuova versione del metodo nella sottoclasse** senza introdurre errori?
3. Nel caso in cui i due override indicati sopra siano corretti dal punto di vista matematico in certi casi, è possibile **farli in Java?**

**La risposta è sì a entrambe le prime due domande.** Alla terza domanda possiamo rispondere di sì per il punto 1, no per il punto 2. Rimandiamo però all'**Appendice** le spiegazioni per chi desidera approfondire questi temi (si noti che l'Appendice **non**

fa parte del materiale di studio per superare l'esame di PPOO).

# Lezione 19

## Interfacce, generici vincolati e alberi di ricerca

In questa lezione intordurremo le *interfacce Java (interface)* e i *generici vincolati*. Per illustrare questi costrutti, intordurremo una versione generica degli alberi binari di ricerca. Per valutare se un albero binario è di ricerca, è necessario che i valori contenuti nei suoi nodi possano essere ordinati secondo un criterio di ordinamento totale. Pertanto, chiederemo che le classi che possono essere sostituite alla variabile di classe T, che sta per il tipo degli elementi dell'albero, contengano valori sui quali è definita una **relazione di ordine, in particolare, una relazione di ordine crescente**.

**Interfacce Java (interface).** Per spiegare la nozione di vincolo su una classe generica T è necessario prima conoscere la nozione di interfaccia in Java. Un'interfaccia Java si presenta come una sequenza di signature (firme) di metodi pubblici non statici **senza nessuna implementazione**. Tuttavia, un'interfaccia **può contenere metodi statici implementati**<sup>17</sup>. Un'interfaccia viene definita usando la parola chiave *interface*, segue un esempio:

```
//Tree.java
public interface Tree {
    public boolean empty();

    public void preOrder();

    public void inOrder();

    public void postOrder();
}
```

Notate che nelle lezioni precedenti abbiamo definito Tree come classe astratta in quanto all'interno della definizione c'erano dei metodi di istanza non astratti, come public String

---

<sup>17</sup> In realtà da Java 8 in poi nelle interfacce si possono trovare dei metodi non statici con body, detti *metodi di default* (<https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html>) Tuttavia, ciò non cambia quanto spiegato in questa lezione.

`toString()`. Nell'esempio di questa lezione, invece, `Tree` definisce solo una serie di firme di metodi, che rappresentano l'interfaccia pubblica di tutte le sue sottoclassi. Ha quindi senso pensare di definire `Tree` come interface, anziché come classe astratta.

Una classe può implementare un'interfaccia: al posto della parola chiave `extends`, usiamo la parola chiave `implements`. **Affinché una classe non abstract implementi una interfaccia I, deve sovrascrivere con metodi concreti tutti i metodi astratti di I.** Segue un esempio:

```
//Leaf.java
public class Leaf implements Tree {
    public boolean empty(){return true;}

    public void preOrder() {}

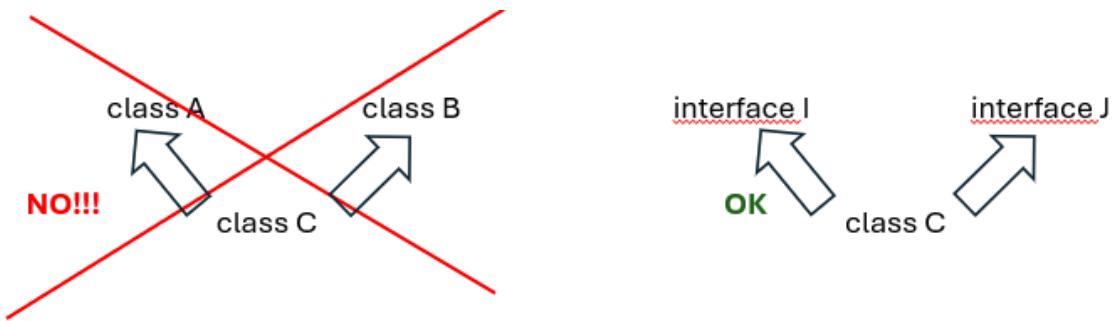
    public void inOrder() {}

    public void postOrder(){}
}
```

Potremmo chiederci quali sono allora le differenze tra un'interfaccia e una classe astratta:

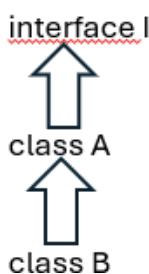
- Una classe astratta può avere degli attributi, dei costruttori e dei metodi non-astratti, un'interfaccia no.
- Inoltre una classe Java può estendere al più una classe (non astratta o astratta), ma può implementare un numero qualunque di interfacce, scrivendo `C implements I, J, ...`.

Notiamo che nel caso delle classi (non astratte e astratte), se per esempio abbiamo due classi A, B che forniscono due versioni concrete diverse dello stesso metodo `m()`, se la classe C estendesse sia A che 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. Infatti, l'implementazione del metodo deve essere presente nella classe C.

**Un esempio di deduzione a partire dalle definizioni date.** Supponiamo di avere B che estende A, la quale implementa l'interfaccia I:



qual è la relazione tra B ed I? A implementa tutti i metodi di I, dunque lo stesso vale per B che estende A. Concludiamo: anche **B implementa I** perché eredita (eventualmente sovrascrivendoli) tutti i metodi di A, tra cui quelli che implementano quelli di I.

**L'interfaccia Comparable<T>.** Le interfacce rappresentano un costrutto potente per mettere in relazione classi appartenenti a diverse librerie. In questo esempio, utilizzeremo l'interfaccia Comparable<T>, predefinita nel linguaggio Java, per costringere il programmatore che sviluppa applicazioni utilizzando alberi binari a impiegare come valori degli elementi dell'albero binario dei valori appartenenti a insiemi ordinati (se no, come si comparano i valori dei nodi, per esempio, per inserire i nodi nella giusta posizione dell'albero?). Per tale scopo, è necessario che le classi che definiscono tali valori contengano un metodo preciso per confrontare tra loro tali argomenti.

Per definire gli alberi di ricerca su una generica classe T, utilizzeremo l'interfaccia **Comparable<T>**. Essa ha unico metodo astratto **int compareTo(T y)**. Questa è la firma di un metodo di confronto tra l'oggetto x e l'oggetto y. Le implementazioni di questo metodo devono seguire la seguente convenzione: se y è in T, e dato il criterio di ordinamento scelto per comparare gli elementi di T, allora x.compareTo(y) dà come risultato:

- **un numero negativo se x precede y nell'ordinamento;**
- **0 se x e y sono uguali;**
- **un numero positivo se x segue y nell'ordinamento.**

Naturalmente, il significato concreto di "precede", "uguale", "segue" dipende dalla semantica della classe in cui si implementa il metodo compareTo(), esattamente come succede per altri metodi come `toString()` e `equals()`. Pertanto, quando dichiariamo che

```
class C implements Comparable<C> {...}
```

siamo tenuti a fornire in C una implementazione per **int compareTo(C y)**. Così facendo, scegliamo una nozione di egualanza e una nozione di ordine per C (il criterio di ordinamento!). Da notare che se definiamo la classe C che implementa l'interface Comparable "stretta a C", questo significa che compareremo istanze di C con istanze di C, e non di altre classi.

Alcuni esempi di `compareTo()` sono presenti nelle seguenti classi di libreria di Java: le classi predefinite **Integer**, **Double**, **String** implementano Comparable<T> rispettivamente per T=Integer, Double, String. Se T=Integer, o T=Double, allora x.compareTo(y) è l'ordine tra numeri. Se T=String, allora x.compareTo(y) è l'ordine lessicografico, in pratica, l'ordine del vocabolario.

**Tipi Generici Vincolati.** Possiamo definire una classe astratta Tree<T> di alberi di ricerca su elementi di tipo generico T aggiungendo un vincolo alla classe generica T che utilizziamo, ovvero possiamo scrivere:

```
interface Tree<T extends I> {...}
```

dove I è un' interfaccia o una classe astratta (bizzarramente, qui si scrive sempre "T extends I" e non "T implements I" se I è un interfaccia).

Allora, quando implementiamo Tree<T>, sappiamo che abbiamo a disposizione tutti i metodi di I, oltre a quelli definiti in Tree. Per esempio, scrivendo

```
interface Tree<T extends Comparable<T>> {...}
```

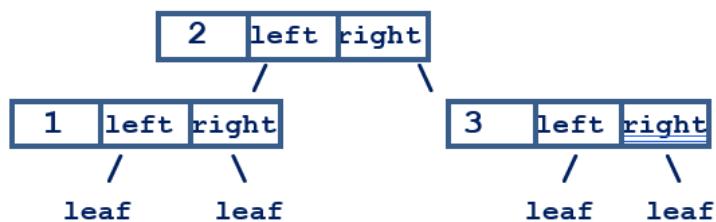
avremo a disposizione il metodo `x.compareTo(y)`. Notate che, in questo modo, abbiamo introdotto un vincolo da rispettare: quando in un'applicazione istanziamo `T` con una classe concreta `C` (volendo un albero `Tree<C>` di elementi di tipo `C` per una particolare classe `C`), possiamo solo scegliere tra quelle classi che estendono `I` (se `I` è una classe astratta) oppure implementano `I` (se `I` è una interfaccia), altrimenti il compilatore 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 degli alberi binari di ricerca i cui elementi sono di tipo generico `T`, abbiamo bisogno di un vincolo su `T`: chiediamo che `T` estenda l'interfaccia `Comparable<T>`, ovvero che sull'insieme `T` sia definita una relazione di ordine e, di conseguenza, esista un metodo di confronto dei suoi elementi, `compareTo()`. In questo modo possiamo usare `compareTo()` per confrontare gli oggetti di `T` nei metodi dell'albero (per esempio nei metodi di aggiunta e di cancellazione di un nodo). Questo implica che quando scegliamo un valore di tipo per `T` dobbiamo scegliere una classe `C` che estenda `Comparable<C>`, cioè che fornisca una implementazione per `int compareTo(C y)`.

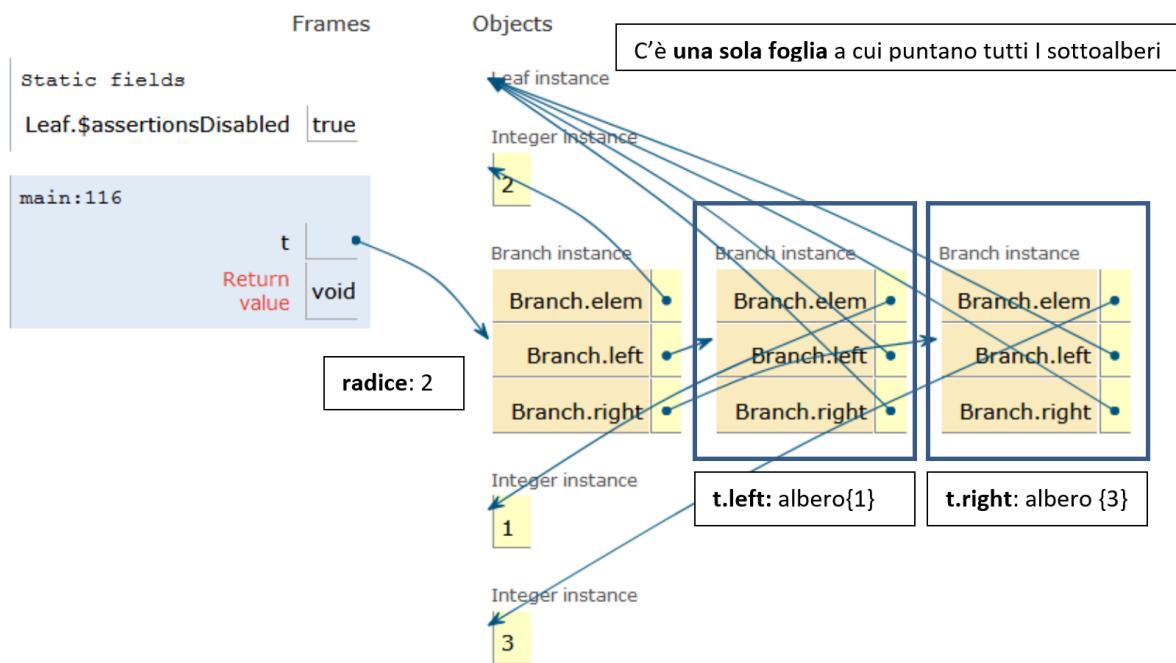
**Un esempio di albero di ricerca.** Ecco una **rappresentazione semplificata** di un albero di ricerca `t={1,2,3}` in Java. Il tipo di `t` è `Tree<T>`, ottenuto istanziando `T=Integer`. L'albero `t` ha radice 2 e figli 1 e 3. La radice di `t` contiene l'elemento 2, e l'indirizzo dei sottoalberi `t.left` 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. Ricordiamo 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 istanze di Integer (più o meno delle "scatole" esterne di contenuto 1,2,3). Inoltre è possibile che gli oggetti di tipo Leaf siano condivisi. In questo particolare esempio, se facessimo la rappresentazione precisa, avremmo un solo oggetto Leaf fisico; inoltre, dai nodi contenenti 1 e 3 partirebbero quattro puntatori che arrivano allo stesso oggetto 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 parametrizzata su una generica classe T, con una relazione di ordine implementata nel metodo `compareTo()` dell'interfaccia `Comparable<T>`. Come abbiamo detto, questa richiesta si traduce nel vincolo: **T extends Comparable<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 e Lezione 18) perché gli

algoritmi per gestire gli alberi binari di ricerca sono gli stessi. Come test per il codice scritto, sceglieremo per T una nuova versione della classe **Contatto** (dalla Lezione 07), versione in cui siamo tenuti a implementare l'interfaccia Comparable<T> con T=Contatto, ovvero il metodo compareTo() per comparare diversi oggetti di tipo Contatto. Potremmo così ottenere anche la classe **Tree<Contatto>** degli alberi di ricerca sugli oggetti di tipo Contatto.

```
// Tree.Java
// Alberi di ricerca con elementi di tipo generico T
// vincolata a estendere l'interfaccia Comparable<T>.
// Questo vuol dire che ogni classe che instanzia T
// dovrà implementare Comparable<T> e quindi contenere 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
    conversione a String del contenuto dell'albero.
    //Non forniamo spiegazioni sul suo funzionamento, non e'
    essenziale

    public String toString()
        {return toStringAux("", "___", "    ", "    ");}
    /* Conversione albero --> stringa. Ogni albero viene
    trascritto in una
    stringa che permette la stampa dall'alto verso il basso, con i
    sottoalberi disegnati piu' a
    destra dell'albero di cui fan parte */
}

//end class Tree
```

```

//Leaf.Java
//ALBERI VUOTTI 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) {

    /* if (x.compareTo(elem) == 0)
       return true;
    else if (x.compareTo(elem) < 0)
        return left.contains(x);
    else
        return right.contains(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 = left.remove(elem);
            //per evitare ripetizioni
            return this;
            // restituisco l'albero modificato
        }
    }

//Metodo che gestisce la parte NON pubblica della
//conversione a stringa.
//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 getName()
    {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 (il metodo
nextDouble() di Random restituisce valori pseudo-random in [0,
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()); //Inserisco i nodi in t
uno 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
destrti");
System.out.println(u);

//Creo un albero u 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 = w.remove(c);
System.out.println(w);

```

```

    }
}

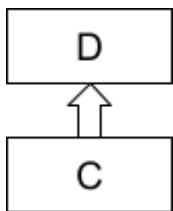
// end class TestTree

```

**Approfondimento sui cast.** Adesso che abbiamo introdotto il concetto di interfaccia (le Java interface, che inducono anch'esse dei tipi, come le classi), possiamo dare la definizione generale di cast (esplicito) corretto. Ricordiamo prima la definizione semplificata che abbiamo dato nella Lezione 14.

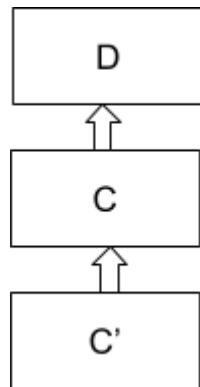
**Definizione semplificata di cast corretto.** Consideriamo il seguente cast al tipo **C** di un oggetto **obj** con tipo apparente **D**: `((C) obj)`.

- **A tempo di compilazione:** dati D tipo apparente di obj e C tipo del cast `((C) obj)` si vuole che  $C <: D$  (ovvero che C sia sottoclasse di D):



Altrimenti il compilatore dà errore.

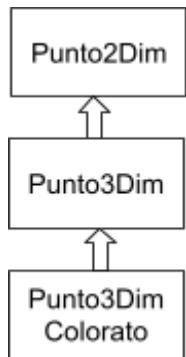
- **A runtime:** un cast `(C) obj` con D tipo apparente di obj non dà errore solo se:
  - obj ha tipo vero C;
  - oppure obj ha tipo vero  $C' <: C$ . Cioè,  $C'$  è una sottoclasse di C. Sappiamo già che  $C <: D$  se si è superata la compilazione, quindi per transitività  $C' <: D$ :



Altrimenti la JVM solleva una *ClassCastException* a runtime e termina il programma, dato che il tipo vero di obj non è confrontabile con il tipo C.

Mostriamo un esempio che illustra la definizione semplificata.  
Date le classi:

```
class Punto2Dim {...}
class Punto3Dim extends Punto2Dim {...}
class Punto3DimColorato extends Punto3Dim {...}
```



possiamo considerare obj di tipo apparente **D = Punto2Dim** e il cast `((Punto3Dim)obj)`, quindi con **C = Punto3Dim**.

Per esempio:

```
Punto2Dim obj = new Punto3Dim(5, 20, 4);
Punto3Dim p3d = (Punto3Dim) obj;
```

Siccome **Punto3Dim <: Punto2Dim**, allora il compilatore non segnala problemi con questo cast.

Se a runtime obj avesse tipo vero **C' = Punto3DimColorato**, allora non si avrebbe errore a runtime, perché **Punto3DimColorato <: Punto3Dim <: Punto2Dim**.

Per esempio:

```
Punto2Dim obj1 = new Punto3DimColorato(5, 20, 4, "blu");
Punto3Dim p3d1 = (Punto3Dim) obj1;
```

Intuitivamente è corretto perché Punto3DimColorato è sottotipo di Punto3Dim, che a sua volta è sottotipo di Punto2Dim. Quindi Punto3DimColorato è sottotipo di Punto2Dim (per transitività), la compilazione ha successo e non ci sono errori a runtime.

Ora introduciamo la definizione generale.

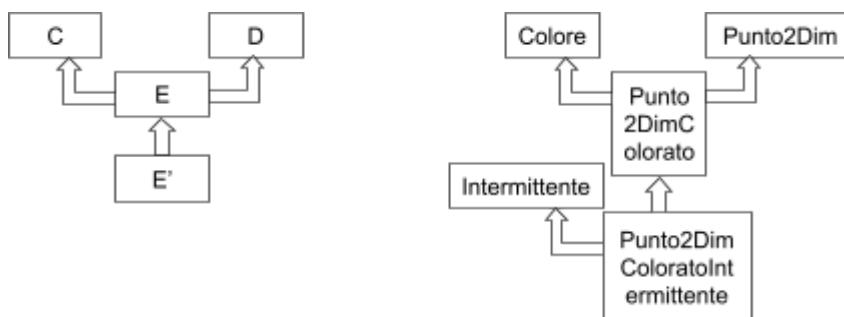
**Definizione generale di cast corretto.** Consideriamo il seguente cast al tipo **C** di un oggetto **obj** con tipo apparente **D: ((C) obj)**.

1. **A tempo di compilazione:**  $((C) \text{ obj})$  con **obj di tipo apparente D** viene considerato di tipo C se esiste un tipo E sottotipo sia di C che di D, ovvero  $E <: C$  e  $E <: D$ . Ricordiamo che o C o D o entrambi devono essere delle interfacce, ovvero solo una tra le due può essere una classe:



Altrimenti il compilatore dà errore.

2. **A runtime:** un cast  $(C)\text{obj}$  con D tipo apparente di obj non dà errore a runtime solo se:
  - a. obj ha tipo vero C;
  - b. oppure obj ha un tipo vero  $E' <: E$ , per cui  $E' <: C$  e  $E' <: D$ , dato il vincolo verificato a tempo di compilazione (il compilatore, infatti, verifica che  $E <: C$  e  $E <: D$ ):



Altrimenti la JVM solleva una `ClassCastException` a runtime e termina il programma, dato che il tipo vero di obj non è confrontabile con il tipo C.

Questa definizione generale implica quella semplificata, basta avere  $E = C$  e  $E' = C'$ . **Nella definizione generale si tiene in conto il fatto che una classe sia sottoclasse di una sola classe** (perché in Java abbiamo solo l'ereditarietà singola, non quella multipla), **ma possa anche implementare una o più**

**interfacce:** da qui la possibilità che esistano **E** ed **E'** sottotipi sia di **C** che di **D**.

Riprendiamo la gerarchia di classi e interfacce introdotta nella definizione sopra, per mostrare un esempio che illustra la definizione generale. Date quindi le classi e le interfacce:

```
<<class>>Punto2Dim      <<interface>>Colore  
  
    \<<ext>>          /<<imp>>  
  
<<class>>Punto2DimColorato      <<interface>>Intermittente  
  
    \<<ext>>          /<<imp>>  
  
<<class>>Punto2DimColoratoIntermittente
```

possiamo considerare obj di tipo apparente **D = Punto2Dim** e il cast **((Colore) obj)**, quindi con **C = Colore**.

```
Punto2Dim obj = new Punto2DimColoratoIntermittente(2, 5,  
"blu", "blink");  
Colore col = ((Colore) obj);
```

Siccome esiste **E = Punto2DimColorato**, con **Punto2DimColorato <: Punto2Dim** e **Punto2DimColorato <: Colore**, allora il compilatore non segnala problemi con questo cast.

Se a runtime obj avesse tipo vero **E' = Punto2DimColoratoIntermittente**, allora non si avrebbe errore a runtime, perchè **Punto2DimColoratoIntermittente <: Punto2DimColorato <: Colore**.

## Lezione 20

### Interfacce Comparable, Iterator e Iterable

**Lezione 20. Parte 1. La classe Bottiglia come implementazione dell'interfaccia Comparable<Bottiglia>.**

Se ci serve comparare oggetti Bottiglia in base a un criterio di ordinamento, possiamo dichiarare che Bottiglia implementa l'interfaccia Comparable<Bottiglia>. In tal caso, dobbiamo aggiungere alla classe Bottiglia un'implementazione del metodo astratto **int compareTo(Bottiglia b)**, per confrontare due oggetti di tipo Bottiglia.

Utilizzare un'interface di libreria, anziché inventarsi un metodo di comparazione proprio, ha dei notevoli vantaggi pratici. In particolare, implementando Comparable<T>, possiamo usare i metodi generici della classe di libreria Arrays: **void Arrays.sort()** e **void Arrays.binarySearch()** per ordinare un array di bottiglie, e per la ricerca binaria in un array ordinato di bottiglie. In generale, questi metodi possono essere applicati a array con elementi di tipo T qualunque, ma a patto che rispetti il seguente vincolo: **T deve implementare Comparable<T> perché nei body di questi metodi, per ordinare e per cercare elementi che sono oggetti di tipo T, i confronti tra gli elementi vengono fatti proprio usando il metodo compareTo(), assumendo che esso sia implementato in ogni C sostituibile a T.**

La classe Arrays appartiene al package **java.util**. Pertanto, per poter utilizzare tali metodi, è necessario che il codice client importi la classe. Lo si fa con questa direttiva: **import java.util.Arrays;** oppure, più comodamente, con **import java.util.\*;** che, ricordiamo, fa importare le classi del package util utilizzate nel codice client.

Potete trovare la definizione dei metodi di Arrays nelle API di Java. Per esempio, per JDK 24: <https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/Arrays.html>.

Ricordiamo che per convenzione il metodo compareTo() deve restituire un int, in particolare restituire 0 se l'oggetto this è uguale all'oggetto 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 client decide come fare il confronto e di conseguenza

l'implementazione del metodo ma, per un buon funzionamento di librerie come Arrays, è necessario rispettare la convenzione sopra.

Nel caso di **int compareTo(Bottiglia b)**, decidiamo di confrontare gli oggetti di tipo Bottiglia solo in base al loro livello, ignorandone la capacità. Nulla ci vieterebbe di fare una scelta diversa, confrontando le bottiglie (anche o solo) in base alla loro capacità.

Nel codice seguente, utilizziamo anche il costrutto **foreach** su array come soluzione alternativa al for: esso permette di iterare su un array, o una qualsiasi altra struttura-dati delle librerie Java, in modo più elegante rispetto al classico for (scopriremo più avanti nel corso di questa lezione qual è il meccanismo che ci sta dietro).

La sintassi è:

```
for ( <tipo> <nome_variabile> : <nome_array> ) <corpo>
```

Dato un **array v di elementi della classe C**, avremo: **for(C c : v){ ...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 il metodo per confrontare bottiglie basato sul solo livello, tramite metodo compareTo().
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 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>):
vedere la spiegazione più avanti in questa lezione.

/* for (int i=0; i<bottiglie.length; i++)
   System.out.println(b[i]); */

// foreach equivalente al for precedente:
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 20. Parte 2. Le interfacce Comparable<E>, Iterable<E> e Iterator<E>. Una classe T che implementa due interfacce.**

**L'interfaccia Iterable<E>.** Riprendiamo ciò che è stato introdotto nella Lezione 15 tramite le classi MiniLinkedList e MiniIterator. In tale lezione, ci eravamo proposti di scrivere un ciclo che attraversi tutti gli elementi di una lista l di elementi di tipo int **senza rendere pubblici gli indirizzi dei nodi all'interno di l**. L'obiettivo era l'information hiding: impedire di accedere da codice esterno direttamente (cioè senza passare dai metodi di MiniLinkedList) ai nodi della lista l. Questo per impedire di invalidare l'invariante di classe; per esempio, inserendo o eliminando un elemento senza aggiornare opportunamente l'attributo size di MiniLinkedList. Per ottenere tale obiettivo, avevamo definito MiniIterator come iteratore di lista ma, ovviamente, tale classe è stata

sviluppata da noi per motivi didattici, senza tenere conto dell'esistenza di librerie Java che già offrono questa funzionalità e che la utilizzano in classi come Arrays.

Riprendiamo la soluzione vista nella Lezione 15, dove avevamo visto la classe MiniLinkedList e la classe MiniIterator, questa volta usando le **interfacce Iterable<E> e Iterator<E>**.



(Si noti che ListExt è la versione "avanzata" di MiniLinkedList e ListIterator è la versione "avanzata" di MiniIterator, della Lezione 15.)

Il vantaggio è duplice:

1. essendo Iterable<E> un tipo generico, lo si può utilizzare in liste di oggetti di tipo definito nel codice client anziché limitarsi a lavorare con liste di interi;
2. il compilatore Java, attraverso le interfacce, può controllare se la costruzione che facciamo è corretta.

Nel codice che segue ci limitiamo però a sfruttare il vantaggio di cui al punto 2., poiché lavoriamo direttamente su una ListExt di int e quindi definiamo **ListExt** come lista di oggetti di tipo Node che implementa l'interfaccia **Iterable<E>**, **con E = Integer**:

```
public class ListExt implements Iterable<Integer> {...}
```

L'interfaccia Iterator<E> specifica la firma del metodo astratto iterator():

|             |            |  |
|-------------|------------|--|
| Iterator<T> | iterator() | Returns an iterator over elements of type T. |
|-------------|------------|--|

In questo modo si impone un vincolo: ListExt deve implementare il metodo **public Iterator<Integer> iterator()**.

Questo metodo crea un oggetto iteratore della classe ListIterator (che implementa Iterator<E> con E = Integer) e gli passa l'indirizzo al primo nodo dell'oggetto ListExt

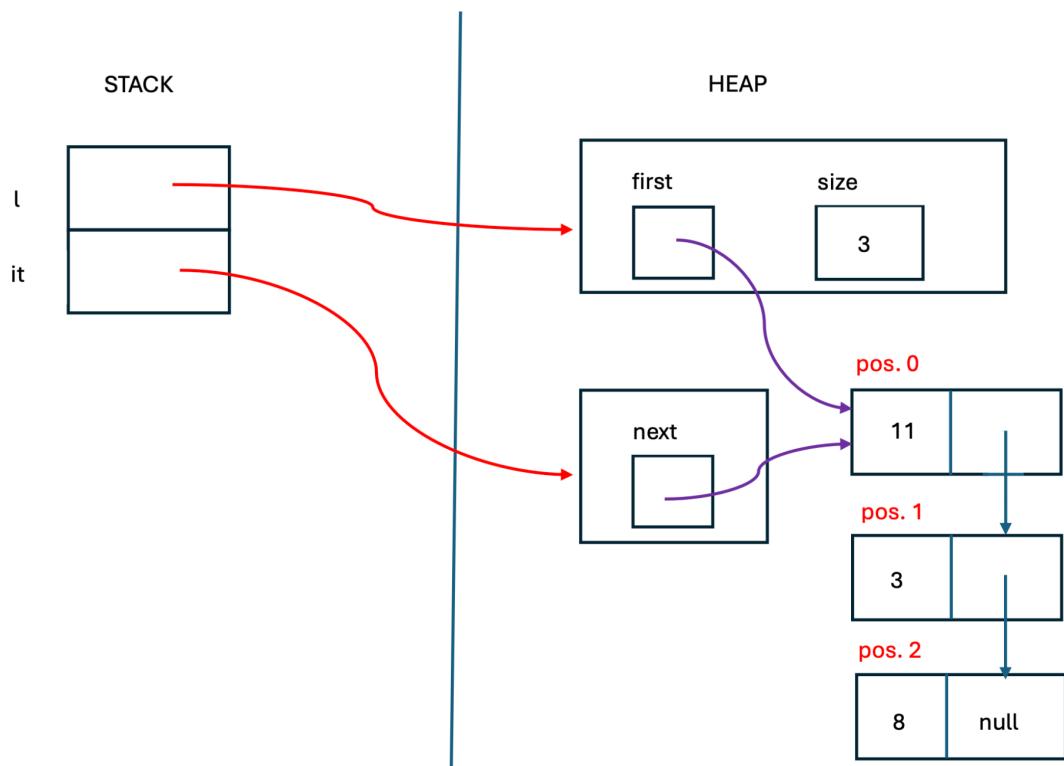
(ovvero il `first` della lista). L'attraversamento della lista verrà fatto tramite l'oggetto iteratore.

```
public Iterator<Integer> iterator(){
    return new ListIterator(first);
}
```

L'interfaccia `Iterator<Integer>` contiene il minimo necessario per eseguire l'attraversamento della lista. In questa lezione ci limitiamo a considerare un campo `next` che contiene il puntatore all'elemento corrente, il metodo `boolean hasNext()`, per sapere quando la traversata di una lista `l` è finita, e il metodo `Integer next()`, per restituire il valore dell'elemento `e` contenuto nel nodo corrente di `l`, e per spostarsi al nodo successivo. Riportiamo parte delle API Java:

|                      |                        |  |
|----------------------|------------------------|--|
| <code>boolean</code> | <code>hasNext()</code> | Returns true if the iteration has more elements. |
| <code>E</code>       | <code>next()</code>    | Returns the next element in the iteration.       |

Dati una `ListExt` `l` e un `ListIterator` `it` creato dal metodo `iterator()` che sta in `ListExt`, possiamo considerare il seguente disegno che illustra un possibile momento d'uso dei due oggetti:



Implementati i metodi richiesti da `Iterable<Integer>` in `ListExt` e da `Iterator<Integer>` in `ListIterator`, possiamo usare il costrutto **`foreach`**: `for (Integer e : l){...e...}`, per `l` di tipo `ListExt`. Notate che, anche se il costrutto si chiama a livello teorico "foreach", nel codice scriviamo "for":

```
ListExt a = new ListExt(); // creo lista vuota
a.add(0, 10); // aggiungo un paio di elementi in testa
a.add(0, 43);
for (Integer o : a) // navigo lista con il foreach
    System.out.println(o);
```

Notate che anche nell'esempio di `ComparaBottiglie` abbiamo usato, in modo implicito, un iteratore quando abbiamo scritto il seguente codice:

```
for (Bottiglia b : bottiglie)
    System.out.println(b);
```

Il **`foreach`** non fa altro che:

- **invocare il metodo `iterator()`** di `ListExt`, che crea un oggetto iteratore (che chiamiamo `it` per semplicità) correttamente inizializzato con il `first` della lista `l`;
- **navigare la lista `l`** (a partire dal `first` e seguendo l'ordine dei suoi elementi fino all'ultimo) e su ciascun elemento eseguire il codice specificato nel body del `for {...e...}`. Per fare ciò, il `foreach` invoca su `it` i metodi `hasNext()` e `next()` di `ListIterator`. Inoltre, il `foreach` lavora senza esporre gli indirizzi degli oggetti `Node` che compongono la lista `l`.

**Questo ci suggerisce che anche la classe `Arrays` di Java implementa `Iterable<E>`**, ecco perché nel codice presentato prima che utilizza array di oggetti di tipo `Bottiglia` abbiamo potuto utilizzare il `foreach`!

Ricordiamo che per poter utilizzare le interfacce `Iterator<E>` e `Iterable<E>`, dobbiamo importare il pacchetto `java.util`. Ricordiamo anche che, volendo gestire una lista di interi, dobbiamo scrivere `Integer` per il tipo degli elementi di `ListExt` e non `int`. Il motivo è che **gli `int` di Java sono tipi primitivi, quindi dobbiamo utilizzare la classe corrispondente `Integer`, perché i tipi generici si possono istanziare solo con tipi classe**. 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. Il compilatore fa la conversione da int e Integer e viceversa in modo automatico con un'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 impone l'implementazione di un metodo compareTo() per confrontare due liste appartenenti alla classe ListExt.

```
public class ListExt implements Iterable<Integer>,
Comparable<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.

Alcuni esempi (riportando le liste in modo semplificato) seguono:

```
{1, 5, 7} < {2, 0}  
{1, 5, 7} < {1, 5, 8}  
{1, 5, 7} = {1, 5, 7}
```

e così via.

```
// Node.java
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 15

/* Ricordiamo com'è fatta l'interfaccia Java Iterable<T>:
   public interface Iterable<T> {
       public Iterator<T> iterator()
   }
 */

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 il first di un oggetto di tipo ListExt
// nel campo next di un
// oggetto di ListIterator, classe che implementa
// Iterator<Integer> (si veda la classe ListIterator sotto).
public Iterator<Integer> iterator(){
    return new ListIterator(first);
}

// 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.*;

public class ListIterator implements Iterator<Integer>{
    // un oggetto di ListIterator contiene un campo che è un
    semplice indirizzo di un nodo:
    private Node next;
    public ListIterator(Node next){this.next = next;}

    /* Per implementare Iterator<Integer> occorre fornire:

```

1. un metodo boolean hasNext() che ci dica se esiste un prossimo oggetto della lista da visitare.
  2. un metodo Integer next() che sposti next sul prossimo oggetto da visitare nella collezione e ne restituisca il valore, un intero.
- Notiamo che il valore originario di next viene perso, quindi la visita si fa una volta sola, per una seconda visita bisogna invocare nuovamente l.iterator() per ottenere un nuovo iteratore. \*/

```

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

public Integer next(){
    int x = next.getElem(); //contenuto del prossimo nodo
    next = next.getNext(); //indirizzo del nodo dopo ancora
    return x;
}
} // end class ListIterator

//TestList.java. Sperimentiamo foreach e compareTo()
//Se usiamo foreach non dobbiamo esplicitamente usare il
metodo 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); //0,1,2,3,...,18,19,20

        System.out.println(" Lista a={0,...,20}" );

        ListIterator it = (ListIterator)a.iterator();
        while (it.hasNext())
            System.out.println(" " + it.next());

        //foreach: non devo usare esplicitamente
        //i metodi come sopra avendo imposto dei vincoli
        //tramite le interfaccie Iterable-Iterator
        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 21

## Eccezioni controllate e non controllate

**Eccezioni.** Un'eccezione in Java è un oggetto della classe **Exception** o di sue sottoclassi che viene usato per segnalare una situazione anomala durante l'esecuzione di un'applicazione. Quando sviluppiamo un programma, possiamo utilizzare le **eccezioni non controllate** per "debuggare" il programma stesso, decidendo se tracciare l'errore lasciando terminare il programma, oppure tracciare l'errore catturando la corrispondente eccezione e quindi proseguendo con la computazione in modo opportuno (tramite l'uso del costrutto try-catch, di cui parleremo tra poco). Una volta corretto il programma, si possono eliminare queste eccezioni.

Invece, è necessario gestire le eccezioni quando le applicazioni non possono essere interrotte immediatamente in condizioni di sicurezza o quando l'errore dipende da condizioni esterne al programma, come la ricezione di dati di input errati, problematiche di rete, etc.

Ci sono classi di eccezioni che descrivono **errori interni** al programma che possono solo essere scoperti a runtime, quindi non tracciabili dal compilatore. 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:

- **ArithmException** (divisione per zero, radice di un numero negativo),
- **IllegalArgumentExcep** (un metodo riceve valori in input non corretti per quel metodo; per esempio, cerchiamo una **parola vuota** in un dizionario italiano),
- **IllegalStateException** (lo stato del programma non consente l'uso di un dato metodo; per esempio, 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).

Definiamo ora dei metodi che vengono accettati dal compilatore ma generano errori a runtime.

//EsempiEccezioni.java

```

/* Scriviamo un metodo che solleva un'eccezione per alcune
situazioni che corrispondono a errori interni a un programma
*/
import java.util.Scanner;

public class EsempiEccezioni {

    public static void test_ArithmeticException()
    {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 invocare un metodo dinamico come toString()
    su null

    public static void test_IllegalArgumentException()
    {
        Scanner obj = new Scanner(System.in);
        System.out.println(obj.nextInt());
    }
    // Se inseriamo un valore non int viene lanciata l'eccezione

    public static void main(String[] args) {
    // Ognuna delle prossime righe solleva una eccezione:
    // scommentatele una per volta.
    // test_ArithmeticException();
    // test_ClassCastException();
    // test_NumberFormatException();
    // test_IndexOutOfBoundsException();
    // test_NullPointerException();
    // test_IllegalArgumentException();
}

```

```
}
```

Ci sono altri tipi di eccezioni, tipicamente causate da **errori esterni** al programma. Per esempio: **FileNotFoundException**, quando si cerca di aprire un file inesistente, oppure **IOException**, quando ci sono errori durante la lettura/scrittura da un file o dalla rete. Per avere a disposizione le classi di eccezioni IO si deve mettere in testa alla parte pubblica del programma una **import java.io.\*;** Un altro esempio di eccezione è quella generata da un comando **assert cond: "messaggio";** che solleva un errore di tipo **AssertionError** quando la condizione cond è falsa (e abbiamo abilitato le asserzioni), terminando così il programma, oltre a inviare un messaggio di errore. In generale, come convenzione, si usano gli errori per indicare situazioni irrecuperabili.

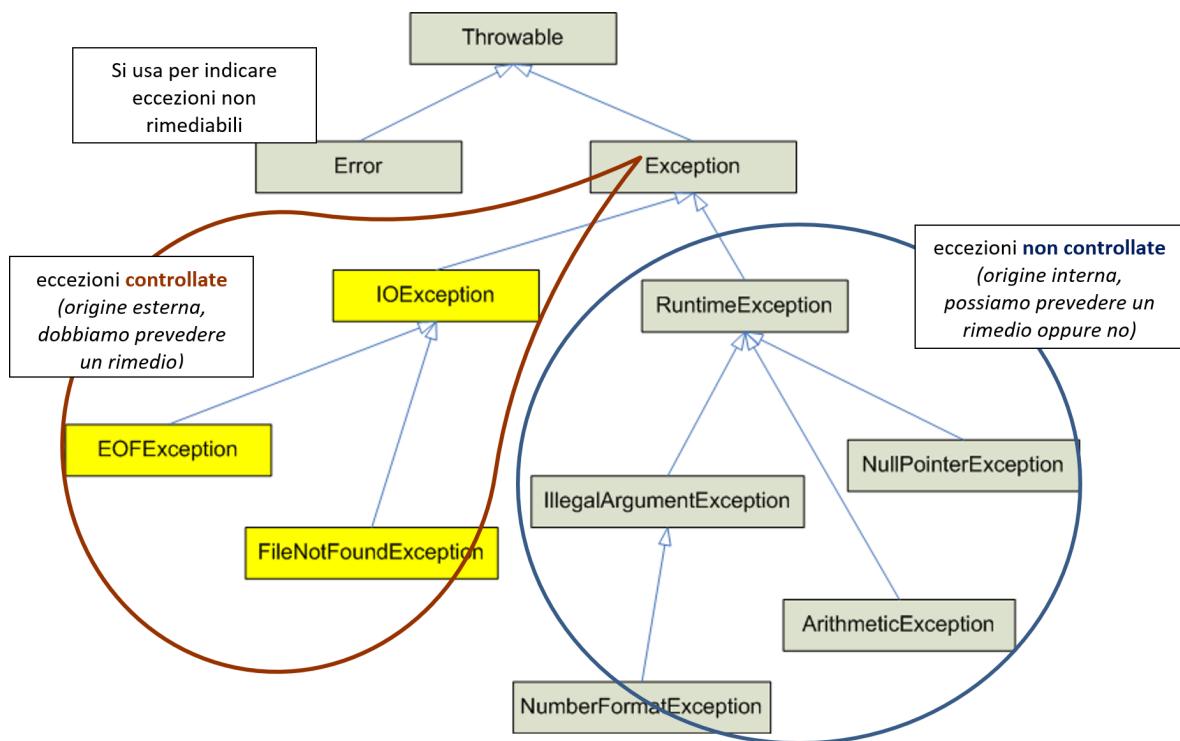
**Eccezioni non controllate e controllate.** Si fa una prima distinzione tra le eccezioni causate da errori interni al programma, che vengono dette **eccezioni non controllate**, e quelle prodotte da cause esterne, che vengono dette **eccezioni controllate**.

- Le **eccezioni non controllate** vengono chiamate così perché è **responsabilità del programmatore decidere se gestirle** quando accadono, prevedendo un rimedio, oppure no. Il programmatore, se non vuole gestire le eccezioni, può limitarsi a cercare tutti gli errori di programmazione che impediscono al programma di non terminare correttamente, evitando la loro comparsa, ma senza prevedere un piano di emergenza se per caso accadessero.
- Invece, il programmatore è tenuto a **gestire sempre le eccezioni controllate** 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. **Il compilatore obbliga a gestire tutte le eccezioni controllate** (che sono un insieme di classi specifico e noto).

Vediamo la gerarchia delle eccezioni predefinite Java. La classe **Eccezione** contiene sottoclassi di eccezioni controllate ("checked"), come **IOException** (input/output Exception), e sottoclassi, come **EOFException**, **FileNotFoundException**. Sempre

la classe Eccezione contiene classi di eccezioni non controllate ("unchecked"), come RunTimeException, e sottoclassi, come NullPointerException. NullPointerException è forse l'eccezione non controllata più comune.

La classe Errore è disgiunta da Eccezione e viene 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, incluse quelle definite da qualsiasi programmatore.



**Cattura di eccezioni.** Non sempre è desiderabile terminare un programma in presenza di un'eccezione e si vorrebbe invece "riparare al problema" e riprendere la normale esecuzione. Per esempio, considerate un programma che riceve da tastiera il nome di un file e deve aprirlo per analizzarne il contenuto. Tale programma non deve necessariamente terminare se il file non esiste. Si può infatti continuare la sua esecuzione gestendo la situazione in modo appropriato, cioè inserendo un ciclo while per: (i) segnalare all'utente il problema; (ii) chiedere di reinserire il nome del file; (iii) leggere il nuovo nome di file inserito dall'utente; (iv) ritentare l'apertura del file. Quando l'utente avrà inserito il nome corretto, l'esecuzione uscirà dal while e si potrà procedere

con l'analisi del contenuto del file. Per fare questo, però, il programma deve poter:

- "catturare" l'eccezione, se si verifica;
- attivare del codice "riparatore" per gestirla.

Per gestire una condizione di errore con del codice appropriato, **le eccezioni possono essere catturate**. Per fare ciò, si scrive il codice che potrebbe sollevare delle eccezioni nella parte (clausola) "try" del seguente costrutto:

```
try{...} catch(TipoEccezione x){...}
```

Per esempio, consideriamo i metodi definiti sopra. Nel main() potremmo gestire l'eccezione di uno di questi metodi nel seguente modo:

```
public static void main(String[] args) {
    try {
        test_NumberFormatException();
    }
    catch (NumberFormatException e) {
        System.out.println("PROBLEMA: " + e.getMessage());
        System.out.println("Qui servirebbe il codice per
rimediare all'eccezione; noi ci limitiamo a visualizzare
a video una notifica del problema");
    }
    System.out.println("Sono uscito dalla catch!!!"); // altro codice del main()
}
}
```

Con il costrutto try-catch si inizia eseguendo il corpo di **try**.

- Se non vengono sollevate eccezioni, viene eseguito tutto il codice nel body della try e si passa all'istruzione successiva al blocco try-catch (nell'esempio, "// altro codice del main()").
- Se invece durante l'esecuzione del codice nella try viene sollevata un oggetto eccezione **t** di tipo **TipoEccezione** (nel nostro esempio, *NumberFormatException*), anziché terminare il programma viene eseguito il corpo della **catch**, con **t** passato come parametro attuale al parametro formale **x**.

Si noti che nella catch abbiamo visualizzato a video il valore restituito da **e.getMessage()**: System.out.println("PROBLEMA: " + e.getMessage()); **le eccezioni hanno infatti un campo message** il cui valore, restituito dal metodo getter, **ci dice quale tipo di problema si è verificato**. Visualizzare a video il valore del message delle eccezioni è importante e si sconsiglia vivamente di NON silenziarle. Per esempio, nel caso delle IOException, il problema potrebbe essere che il file non esiste, che si tenta di scrivere in un file aperto solo in lettura, etc. Nella clausola catch dei costrutti è quindi molto utile visualizzare a video il valore del campo message delle eccezioni per saperne di più!

Finora abbiamo visto esempi in cui le eccezioni vengono lanciate dal codice delle classi di libreria di Java. Tuttavia, anche noi possiamo scrivere codice che lancia eccezioni per segnalare le problematiche che si incontrano durante la sua esecuzione. Come esempio, riconsideriamo la classe Bottiglia. In tale implementazione, anziché usare le asserzioni, potremmo voler far generare un'eccezione se il codice client cerca di aggiungere una quantità negativa di liquido in una bottiglia. **Il campo message delle eccezioni viene inizializzato all'atto della new dell'eccezione** ("throw new IllegalArgumentException("aggiungi: quantita negativa");"):

```
public class Bottiglia {
    //...
    public void aggiungi(int quantita) {
        if (quantita < 0)
            throw new IllegalArgumentException("aggiungi: quantita negativa");
        livello = Math.min(livello + quantita, capacita);
    }
    //...
}
```

Se nella firma di aggiungi() non dichiariamo che può generare l'IllegalArgumentException (come nella definizione sopra), il codice che invoca il metodo può gestire l'eccezione per non interrompere la propria esecuzione in caso si verifichi; come nel metodo main() qui sotto:

```
public static void main(String[] args) {
    Bottiglia b = new Bottiglia(5);
```

```

        int q = ... // la quantità di liquido da aggiungere
        try { b.aggiungi(q);}
        catch (IllegalArgumentException e)
            {System.out.println(e.getMessage());}
    }
}

```

Ovviamente, se il codice che invoca aggiungi() non cattura l'eccezione, verrà bloccata la sua esecuzione in caso aggiungi() la lanci.

In un metodo il cui body può generare eccezioni non controllate non siamo obbligati a dichiarare nella firma che essi possono lanciarle. Diverso è per le eccezioni controllate: in quel caso siamo obbligati ad aggiungere alla firma dei metodi la dichiarazione. **Per indicare che un metodo m() può sollevare un'eccezione di tipo TipoEccezione** durante l'esecuzione del suo body, si può aggiungere alla firma del metodo una clausola **throws** c, scrivendo:

**<tipo\_di ritorno> m() throws TipoEccezione.**

**Quando un metodo m() dichiara la throws di un tipo di eccezione, TipoEccezione, una chiamata a m() può essere presente solo all'interno di un'istruzione try che cattura l'eccezione TipoEccezione.**

Supponiamo che m() sollevi un'eccezione TipoEccezione (anche direttamente, con un'istruzione **throw new TipoEccezione(...)**: attenzione a non confonderla con la 'throws' che compare nelle signature!). Ripetiamo le regole per throws:

1. Nel caso TipoEccezione sia una eccezione **non controllata**, non si è obbligati a scrivere la clausola throws nella firma di m().
2. Nel caso TipoEccezione sia una eccezione **controllata**, si è obbligati a scrivere la clausola throws TipoEccezione nella firma di m().

Osserviamo anche che nella clausola catch dovete catturare tipi di eccezioni che "corrispondono" a quanto può essere generato dal codice interno alla try (ovvero, TipoEccezione o sottotipi). Se dichiarate tipi di eccezioni che non hanno nulla a che vedere con gli oggetti eccezione che possono essere creati all'interno della try, la vostra catch non serve a nulla. Esempio:

**try {**

```

        // codice che può generare una IOException
    }
    catch (NullPointerException e) {...} // NO! non state
catturando la possibile IOException!!

```

Vediamo ora alcuni esempi 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
chiamate a m() in un costrutto "try-catch".
    {throw new Error( "error" );}
//Viene chiamato il costruttore della classe Error, che prende
//un parametro String: ricordate, 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
possibili
            {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

```

**Clausole catch multiple.** Possiamo catturare più eccezioni con il costrutto:

```
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 ne sia sopratipo). Definire catch multiple è utile per poter gestire in modo appropriato i diversi tipi di errore che il codice nella try potrebbe generare. Per esempio, diverso è se non si trova un file dal fatto che ci sia un problema di rete; pertanto, in caso il codice nella try li potesse generare entrambi, è conveniente avere due gestori delle eccezioni separati, uno per ciascun tipo di eccezione.

**Esempi di eccezioni non controllate (cattura possibile, non necessaria).** Come 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 06), 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 firma 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
        // l'eccezione fa terminare il programma.
        System.out.println
            ("\nLa prossima istruzione fa terminare il programma\n");
        new Bottiglia(-5);
    }
}

```

Un esempio simile al precedente è la classe Stack degli stack di interi (Lezione 05). Anche in questo caso sostituiamo l'assert con il lancio di una eccezione non controllata. Essendo l'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 firma 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];
    }
}

```

#### Definizione di nuove classi di eccezioni.

Possiamo definire noi stessi delle nuove classi di eccezioni per rappresentare situazioni anomale che ci pare non vengano ben rappresentate dalle classi di eccezioni di libreria. Per esempio, possiamo definire **MiaEccezione** estendendo una qualunque classe di eccezioni esistente: per le classi di eccezioni create da noi valgono le stesse regole che per le eccezioni di Java. Per esempio, in un'app che gestisce un gioco, si potrebbe creare un'eccezione BadGameException per

creare un nuovo tipo di eccezioni che rappresenti una violazione delle regole da parte del giocatore che sta tentando una mossa vietata:

```
public class BadGameException extends Exception {...}
```

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 firma del metodo in cui la lanciamo con un "throws". Se lo facciamo, siamo obbligati a catturarla con un blocco 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 del toString().
//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 18 sugli alberi binari di ricerca e definire il metodo max() in tutti.

#### **Esempi di eccezioni controllate (cattura necessaria).**

Proviamo a definire un'eccezione controllata, per impedire la creazione di una frazione di denominatore  $\leq 0$ , ma senza far cadere il programma quando questo capita. Per le eccezioni controllate siamo obbligati a scrivere la clausola `throws` nelle segnature dei metodi che le lanciano, e poi a catturarle nel codice che usa quei metodi. Definiamo quindi una classe di eccezioni controllate come sottoclasse della classe IOException di eccezioni controllate. Mettiamo come parametro dell'eccezione il denominatore della frazione rifiutata.

```

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

public 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' un'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

import java.util.*;
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: il compilatore dice "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);
}
}

```

### Clausole catch ripetute.

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( "NON sarò mai eseguita!");
        }
        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 SEMPRE, sia che arriviamo dalla try
    che da un catch
    System.out.println( "\n...posto per la clausola
finale...\n");
}
}
} //end class TestTryCatch
```

## Lezione 22

### Cenni alla progettazione orientata agli oggetti: i design pattern Singleton, Composite, State

(Introduzione liberamente tratta da  
[https://it.wikipedia.org/wiki/Design\\_pattern](https://it.wikipedia.org/wiki/Design_pattern) e integrata)

*Design pattern*, in informatica e specialmente nell'ambito dell'ingegneria del software, un design pattern è un concetto che può essere definito "una soluzione progettuale generale ad un problema ricorrente". Si tratta di una descrizione o modello logico da applicare per la risoluzione di un problema che può presentarsi in diverse situazioni durante le fasi di progettazione e sviluppo del software, ancor prima della definizione dell'algoritmo risolutivo.

Un design pattern è costituito da:

- **il nome**, formato da una o due parole che siano il più possibile rappresentative del pattern stesso;
- **il problema**, ovvero la descrizione della situazione alla quale si può applicare il pattern. Può comprendere la descrizione di classi o di problemi di progettazione specifici, come anche una lista di condizioni perché sia necessario l'utilizzo del pattern;
- **la soluzione**, che descrive gli elementi costitutivi del progetto con le relazioni e relative implicazioni, senza però addentrarsi in una specifica implementazione. Il concetto è di **presentare un problema astratto e la relativa configurazione di elementi adatta a risolverlo**;
- **le conseguenze**, i risultati e i vincoli che derivano dall'applicazione del pattern. Sono fondamentali in quanto possono essere l'ago della bilancia nella scelta dei pattern: le conseguenze comprendono considerazioni di tempo e di spazio, possono descrivere implicazioni del pattern con alcuni linguaggi di programmazione e l'impatto sul resto del progetto.

Studierete questo argomento in modo estensivo nell'insegnamento Sviluppo delle Applicazioni Software del terzo anno. Per ora vedremo tre esempi di design pattern che fatto parte dei design pattern della Gang Of Four (GoF), tra i più noti e utilizzati.

I design pattern GoF si dividono in tre gruppi:

- I pattern **creazionali**, che risolvono problematiche inerenti all'istanziazione degli oggetti. Di questa categoria vedremo il pattern **Singleton**, che ha lo scopo di assicurare che di una classe possa essere creata una e una sola istanza.
- I pattern **strutturali**, che risolvono problematiche inerenti alla struttura delle classi e degli oggetti. Di questa categoria vedremo il pattern **Composite**, utilizzato per manipolare una gerarchia di oggetti in modo uniforme.
- I pattern **comportamentali**, che forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti. Di questa categoria vedremo il pattern **State**, che permette ad un oggetto di cambiare il suo comportamento al cambiare di un suo stato interno.

**Pattern Singleton (liberamente tratto da [https://it.wikipedia.org/wiki/Singleton\\_\(informatica\)](https://it.wikipedia.org/wiki/Singleton_(informatica)) e integrato).**

Il Singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza. L'implementazione più semplice di questo pattern prevede che la classe *singleton* abbia un unico costruttore privato, in modo da impedire l'istanziazione diretta della classe. La classe fornisce inoltre un metodo *getter statico* che restituisce l'istanza della classe (sempre la stessa):

- creandola preventivamente (non studiamo questo approccio)
  - o alla prima chiamata del metodo *get()*

e memorizzandone il riferimento in un attributo privato anch'esso statico. Il secondo approccio si può classificare come basato sul principio della lazy initialization (lett. "inizializzazione pigra") in quanto la creazione dell'istanza della classe viene rimandata nel tempo e messa in atto solo quando ciò diventa strettamente necessario (al primo tentativo di uso, utile perché se non viene mai utilizzata si risparmiano risorse di memoria). Segue un frammento di codice Java in cui si ha proprio l'inizializzazione lazy.

```

public class MioSingolo {

    private static MioSingolo istanza = null;
    //La variabile statica 'istanza' fa da contenitore
    dell'unica istanza della classe MioSingolo; fino a che nessun
    codice esterno ha bisogno dell'oggetto di questa classe, il
    contenuto di istanza sarà null

    //Il costruttore private impedisce la creazione diretta di
    oggetti da parte di classi esterne senza passare dal metodo
    getMioSingolo() che segue
    private MioSingolo() {}

    // Metodo della classe impiegato per accedere al singleton
    public static MioSingolo getMioSingolo() {
        if (istanza == null) { //se vale questo, occorre
        creare l'oggetto, mai stato creato prima
            istanza = new MioSingolo();
        }
        return istanza; //viene restituito sempre lo stesso
        oggetto
    }
}

```

Si noti che qualsiasi classe può essere trasformata in una classe Singleton: basta dichiarare il costruttore della classe come private e aggiungere (i) la variabile (contenitore) statica dell'unica istanza e (ii) il metodo statico per la sua creazione (una sola) e restituzione.

Possiamo immaginare varianti del pattern Singleton, per esempio un pattern Multiton in cui si possono creare al massimo n istanze, memorizzate in una struttura apposita, e poi si usano sempre quelle senza creare altri oggetti, scegliendo tra loro con una politica predefinita.

Nel contesto di questo insegnamento, è interessante studiare il Singleton per vedere un utilizzo elegante di una variabile statica e di un corrispondente metodo accessorio statico, nonché dell'utilità di poter dichiarare un costruttore con visibilità private. In altri insegnamenti, come in Programmazione III, potrete vedere che il pattern Singleton è molto utile per progettare classi che devono fornire un unico punto di accesso a una risorsa, hardware o software. Inoltre, si studieranno i limiti dell'implementazione sopra presentata

in un contesto in cui ci sono più processi in esecuzione (programmazione *multithreading*) e come superarli.

Riportiamo nel seguito un esempio completo di applicazione che fa uso del pattern singleton per utilizzare un UNICO database che più classi utilizzano per offrire tipi di servizi differenti.

```
public class Product {
    private String brand;
    private String model;

    Product(String k, String v) {brand = k; model = v;}
    public String toString() {return brand + ": " + model;}
}

public class Database {
    private static Database instance;
    private Product[] cars = new Product[3];

    private Database() { // database content
        cars[0] = new Product("Toyota", "Yaris");
        cars[1] = new Product("Toyota", "Rav4");
        cars[2] = new Product("FIAT", "Pandina");
    }

    public static Database getInstance() {
        if (instance == null) instance = new Database();
        return instance;
    }

    public int countElements() { return cars.length; }

    public void print() {
        for (Product c : cars) System.out.println(c);
    }
}

public class SimpleSingletonTest {
    public static void main(String[] args) {
        UserInterface.showCatalog();
        Utilities.statistics();
        // l'applicazione utilizza UserInterface e
        Utilities, entrambe configurate per lavorare con l'istanza
        singola di Database.
    }
}
```

```

        // Il Database e' un modello di dati comune alle due
        // classi UserInterface e Utilities, che lo condividono per
        // operare su una base di dati unica.
    }

}

class UserInterface {
    public static void showCatalog() {
        Database catalog = Database.getInstance();
        System.out.println("catalog ID: " +
catalog.hashCode()); // stampo l'ID dell'oggetto
        catalog.print();
    }
}

// Le visualizzazioni a video "System.out.println("catalog ID:
// " + catalog.hashCode());" riportate nei metodi ci servono per
// provare che si sta lavorando su una stessa istanza di
// Database.

class Utilities {
    public static void statistics() {
        Database cars = Database.getInstance();
        System.out.println("catalog ID: " +
cars.hashCode());
        System.out.println("Number of products: " +
cars.countElements()); // stampo l'ID dell'oggetto
    }
}

```

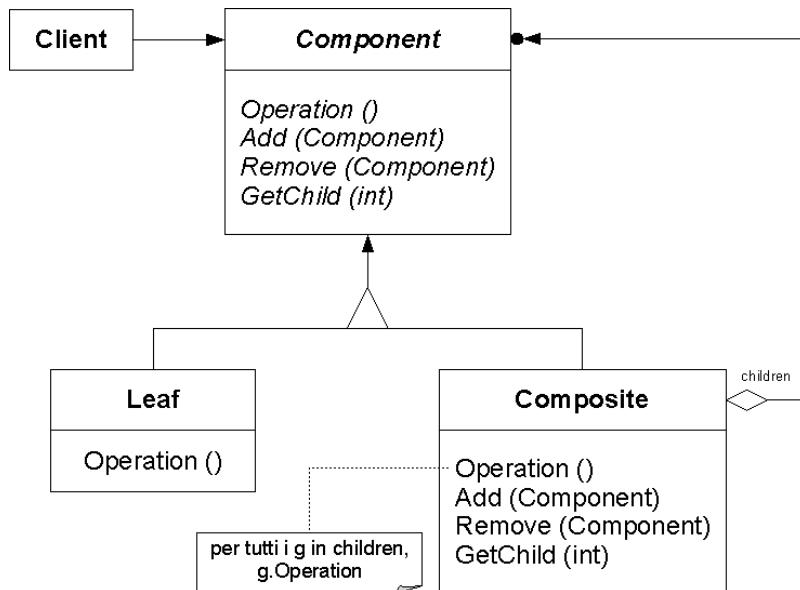
**Pattern Composite (liberamente tratto da  
<https://it.wikipedia.org/wiki/Composite> e integrato).**

Il pattern Composite è un pattern strutturale che permette di trattare un gruppo di oggetti come se fossero l'istanza di un oggetto singolo. Il Composite organizza gli oggetti in una struttura ad albero, nella quale i nodi sono delle composite e le foglie sono oggetti semplici. È utilizzato per dare la possibilità ai client di manipolare oggetti singoli e composizioni in modo uniforme, cioè utilizzando metodi con le stesse firme, indipendentemente dal fatto che si stia invocando un metodo su un oggetto foglia o su un composite che aggredisce oggetti più semplici.

**Scopo del Composite.** Quando i programmatore trattano dati strutturati ad albero devono spesso discriminare se stanno

visitando un nodo o una foglia. Questa differenza è una possibile fonte di complessità per il codice e, se non trattata a dovere, rende il programma facilmente soggetto a errori. La soluzione è adottare un'interfaccia (Java interface) che permetta di trattare oggetti complessi e primitivi in modo uniforme. Il concetto fondamentale è che il programmatore manipola ogni oggetto dell'insieme dato nello stesso modo, sia esso un raggruppamento di oggetti o un oggetto singolo. Le operazioni che possono essere effettuate sul Composite hanno spesso un denominatore comune, per esempio: se il programmatore deve visualizzare a schermo un insieme di figure potrebbe essergli utile definirne il ridimensionamento in modo da avere lo stesso effetto del ridimensionamento di una singola figura.

### Struttura del Composite

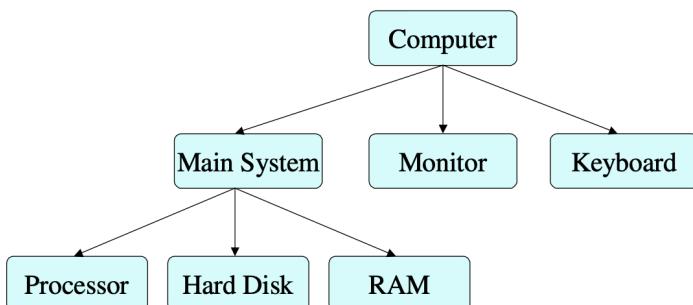


- *Client*: manipola gli oggetti attraverso i metodi dell'interfaccia Component.
- *Component*: dichiara l'interfaccia degli oggetti nella composizione, siano essi oggetti primitivi o oggetti composti, elencando le operazioni per l'accesso e la manipolazione agli oggetti stessi.
- *Composite*: definisce il comportamento per i componenti aventi componenti/figli, salva i figli e implementa le operazioni ad essi connessi nell'interfaccia Component.
- *Leaf*: definisce il comportamento degli oggetti primitivi.

Attraverso l'interfaccia Component, il Client interagisce con gli oggetti della gerarchia. Se l'oggetto desiderato è una Leaf, la richiesta è processata direttamente; altrimenti, se è una Composite, viene rimandata ai figli.

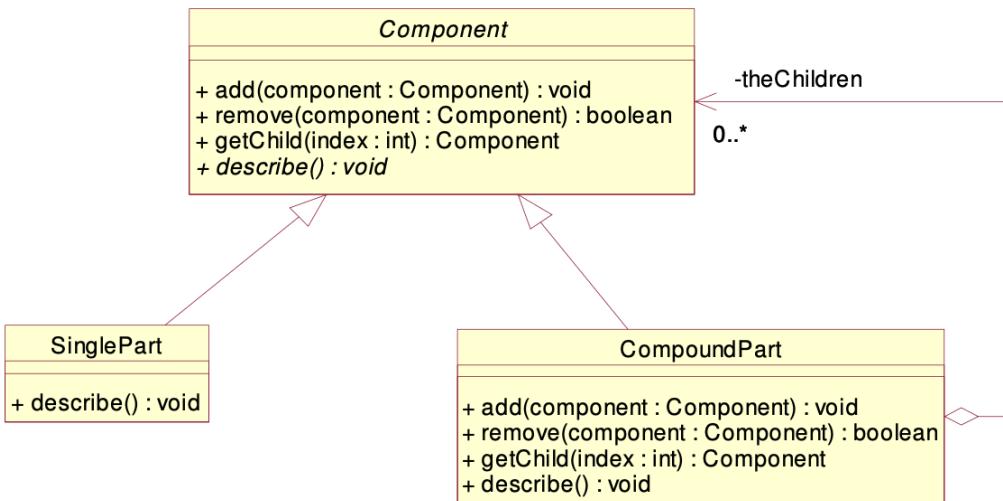
**Esempio di applicazione del pattern Composite (adattato a partire dall'esempio di Composite delle dispense "GoF's Design Patterns in Java", di Franco GUIDI POLANCO, Politecnico di Torino, 2002).**

Nel magazzino di una ditta fornitrice di computer ci sono diversi prodotti, quali computer pronti per la consegna, e pezzi di ricambio (o pezzi destinati alla costruzione di nuovi computer). Dal punto di vista della gestione del magazzino, alcuni di questi pezzi sono pezzi singoli (indivisibili), altri sono pezzi composti da altri pezzi. Il monitor, la tastiera (keyboard), la RAM, il disco rigido (Hard Disk) e il processore (Processor) sono pezzi singoli. Invece il main system è un pezzo composto da tre pezzi singoli (processore, disco rigido e RAM). Un altro esempio di pezzo composto è il computer, che a sua volta è composto da un pezzo composto (il main system) e da due pezzi singoli (monitor e tastiera), come da figura.



Il problema che vogliamo risolvere è rappresentare e gestire in modo omogeneo tutti gli elementi presenti nel magazzino, sia i singoli componenti, sia quelli composti da altri componenti.

Possiamo schematizzare la soluzione che utilizza il pattern nel seguente modo:



Mostramo ora un'implementazione di questo esempio in Java.

Abbiamo una classe astratta Component che definisce l'insieme dei metodi comuni agli oggetti singoli e composti, e implementa alcune operazioni come metodi concreti i quali offrono un'implementazione di default, che vada quindi bene sia per gli oggetti di SinglePart, che per gli oggetti di CompoundPart. In particolare, queste operazioni sono:

- La `add(Component c)` e la `remove(Component c)`, che sollevano un'eccezione del tipo `SinglePartException` se vengono invocate su un oggetto `SinglePart`.
- La `getChild(int n)`, che restituisce null (questa è stata una scelta di progettazione, un'altra possibilità era sollevare anche in questo caso un'eccezione).

Invece il metodo `describe()` è dichiarato come metodo astratto, da implementare in modo opportuno in ciascuna delle sottoclassi. Si noti che il costruttore di `Component` riceve una stringa contenente il nome del componente, che verrà assegnato ad ognuno di essi.

```

public abstract class Component {
    public String name;
    public Component(String aName) {
        name = aName;
    }
    public abstract void describe();

    public void add(Component c) throws SinglePartException {
        if (this instanceof SinglePart)
    }
}
  
```

```

        throw new SinglePartException( );
    }

public void remove(Component c) throws
SinglePartException {
    if (this instanceof SinglePart)
        throw new SinglePartException( );
}

public Component getChild(int n){
    return null;
}
}

```

La classe **SinglePart** estende la classe **Component**. Possiede un costruttore che consente l'assegnazione del nome del singolo componente, il quale verrà memorizzato tramite l'invocazione al costruttore della superclasse. La classe SinglerPart fornisce anche l'implementazione del metodo describe().

```

public class SinglePart extends Component {
    public SinglePart(String aName) {
        super(aName);
    }
    public void describe(){
        System.out.println( "Component: " + name );
    }
}

```

Anche la classe CompoundPart estende Component, e fa l'override sia dei metodi di gestione dei componenti (add(), remove(), getChild()), sia del metodo describe(). Si noti che il metodo describe() visualizza a video in primo luogo il nome dell'oggetto composto corrente e poi scorre l'elenco dei suoi componenti, invocando il metodo describe() di ognuno di essi tramite binding dinamico. Il risultato sarà tale che, insieme alla stampa del nome dell'oggetto composto, verranno anche visualizzati i nomi dei suoi componenti.

```

import java.util.Vector;

public class CompoundPart extends Component {
    private Vector<Component> children;

```

```

//struttura-dati che sta tra le Collections di Java:
https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java.util/Vector.html

public CompoundPart(String aName) {
    super(aName);
    children = new Vector<Component>();
}

public void describe(){
    System.out.println("Component: " + name);
    System.out.println("Composed by:");
    System.out.println("{");
    int vLength = children.size(); //metodo di Vector
    for( int i=0; i< vLength ; i ++ ) {
        Component c = children.get( i );
        c.describe();
    }
    System.out.println("}");
}

public void add(Component c) throws SinglePartException {
    children.addElement(c); //metodo di Vector
}

public void remove(Component c) throws SinglePartException{
    children.removeElement(c); //metodo di Vector
}

public Component getChild(int n) {
    return children.elementAt(n); //metodo di Vector
}
}

```

Si noti che in questa implementazione ogni oggetto **CompoundPart** memorizza i propri componenti in un **Vector**.

La classe **SinglePartException** che segue rappresenta l'eccezione (non controllata) che verrà sollevata nel caso che le operazioni di gestione dei componenti (**add()**, **remove()**, **getChild()**), vengano invocate su una parte singola.

```

class SinglePartException extends Exception {
    public SinglePartException( ){

```

```

        super( "Not supported method" );
    }
}

```

L'applicazione **CompositeExample** fa le veci del codice client e **gestisce i diversi tipi di componenti, tramite l'interfaccia comune fornita dalla classe Component**. Nella prima parte dell'esecuzione si creano dei componenti singoli (monitor, keyboard, processor, ram e hardDisk), dopodiché viene creato un oggetto composto (mainSystem) con tre di questi oggetti singoli. L'oggetto composto appena creato serve, a sua volta, per creare, insieme ad altri pezzi singoli, un nuovo oggetto composto (computer). L'applicazione invoca poi il metodo `describe()` su un oggetto singolo, sull'oggetto composto soltanto da componenti singoli, e sull'oggetto composto da componenti singoli e componenti composti. Infine prova a fare un tentativo di aggiungere un componente ad un oggetto corrispondente a un pezzo singolo (operazione che a runtime lancerà un'eccezione).

```

public class CompositeExample {
    public static void main(String[] args) {
        // Creates single parts
        Component monitor = new SinglePart("LCD Monitor");
        Component keyboard = new SinglePart("Italian Keyboard");
        Component processor = new SinglePart("Pentium III
Processor");
        Component ram = new SinglePart("256 KB RAM");
        Component hardDisk = new SinglePart("40 Gb Hard Disk");
        // A composite with 3 leaves
        Component mainSystem = new CompoundPart( "Main System" );
        try {
            mainSystem.add( processor );
            mainSystem.add( ram );
            mainSystem.add( hardDisk );
        }
        catch (SinglePartException e){
            e.printStackTrace();
        }
        // A Composite compound by another Composite and one Leaf
        Component computer = new CompoundPart("Computer");
        try{
            computer.add( monitor );

```

```

computer.add( keyboard );
computer.add( mainSystem );
}
catch (SinglePartException e){
    e.printStackTrace();
}
System.out.println("**Tries to describe the 'monitor' component");
monitor.describe();
System.out.println("**Tries to describe the 'main system' component"
);
mainSystem.describe();
System.out.println("**Tries to describe the 'computer' component" );
computer.describe();
// Wrong: invocation of add() on a Leaf
System.out.println( "***Tries to add a component to a single part (leaf) " );
try{
    monitor.add( mainSystem );
}
catch (SinglePartException e){
    e.printStackTrace();
}
}
}

```

**Qualche osservazione sul Composite.** Nell'implementazione presentata, la classe astratta Component fornisce un'implementazione di default per i metodi di gestione dei componenti (add(), remove(), getChild()). Dal punto di vista del *Composite pattern*, sarebbe anche valida la dichiarazione di questi metodi come metodi astratti, lasciando l'implementazione alle classi SinglePart e CompoundPart.

In questa lezione, non ci soffermiamo su questa implementazione alternativa, tuttavia pensiamo sia importante accennarne poiché una descrizione completa di un pattern include eventuali soluzioni alternative e eventuali svantaggi e limiti di utilizzo.

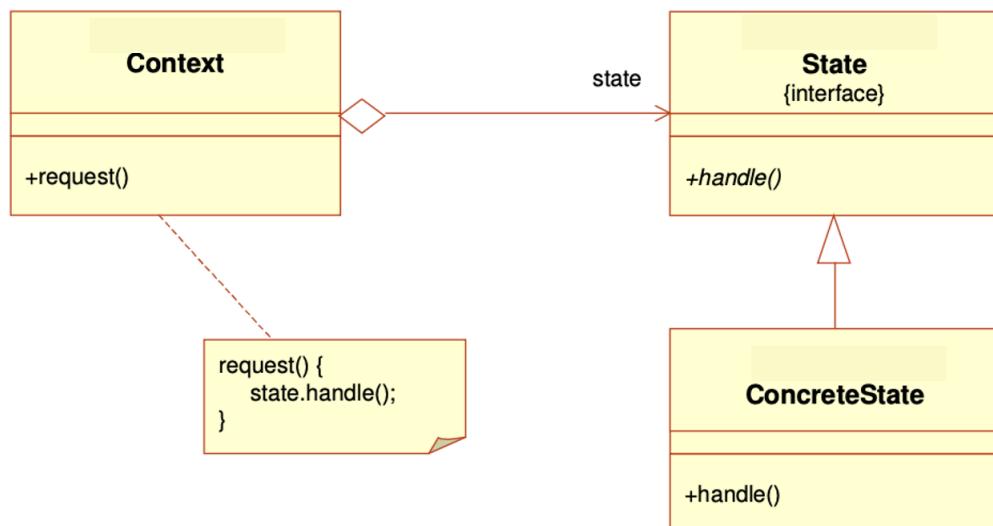
Per quanto riguarda il Composite, il limite principale è che le operazioni sulle componenti singole e composte devono essere della stessa natura, per poterle trattare in modo uniforme.

## Pattern State

Il pattern State è un pattern comportamentale. Permette di far dipendere il comportamento di un oggetto dallo stato in cui si trova l'oggetto stesso.

**Scopo dello State.** Consente a un oggetto di modificare il suo comportamento a runtime quando il suo stato interno cambia. Anche il suo stato viene implementato come un oggetto, in modo da poter avere la massima flessibilità in termini di memorizzazione di dati utili (nei campi dell'oggetto che rappresenta lo stato) e di funzionalità (metodi dell'oggetto che rappresenta stato). Questo meccanismo sottostante allo State si chiama comunemente *composizione* di oggetti.

## Struttura dello State



La classe **Context** rappresenta l'insieme degli oggetti che cambiano comportamento al variare del loro stato interno, contenuto in un opportuno campo `state` e implementato come oggetto di tipo `State`. **L'interfaccia State<sup>18</sup> è soprattutto di un numero variabile di classi `ConcreteState`, rappresentanti ciascuna uno stato in cui si può trovare l'oggetto `Context`.** Nel campo `state` ci dovrà quindi sempre essere un puntatore all'oggetto `State` che rappresenta lo stato corrente dell'oggetto `Context` stesso. **L'esecuzione di ogni "request"** (invocazione del metodo `request()`) **fatta all'oggetto `Context` viene delegata allo state** (si veda nel diagramma UML sopra),

<sup>18</sup> **State può anche essere una classe astratta**, dipende dall'applicazione del pattern.

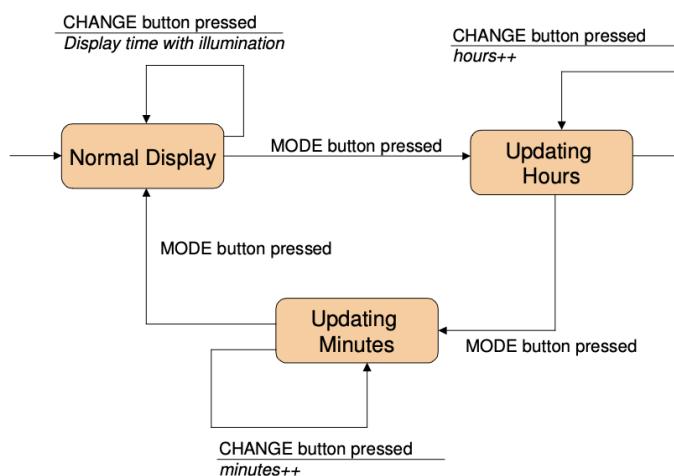
in modo che il comportamento dell'oggetto Context cambi in funzione dell'oggetto State presente nel suo campo state.

**Esempio di applicazione del pattern State (adattato a partire dall'esempio di State delle dispense "GoF's Design Patterns in Java", di Franco GUIDI POLANCO, Politecnico di Torino, 2002).**

Vediamo come caso di studio un'astrazione di un semplice orologio digitale i cui pulsanti sono abilitati a espletare funzioni differenti a seconda della modalità (ovvero lo stato) in cui si trova l'orologio. Si pensi ad un orologio che possiede due pulsanti: MODE e CHANGE. Il primo pulsante serve per settare il modo di operazione di tre modi possibili: "visualizzazione normale", "modifica delle ore", "modifica dei minuti". Imvece il secondo pulsante:

- serve per accendere la luce del display, se l'orologio è in modalità di visualizzazione normale,
- oppure per incrementare di una unità le ore, se è in modifica di ore,
- o per incrementare di un'unità i minuti, se è in modalità di modifica di minuti.

Il seguente diagramma di stati serve a rappresentare il comportamento dell'orologio:



Un approccio semplicistico conduce all'implementazione del codice di ogni operazione come una serie di test, come appare in questo frammento di pseudocodice:

```

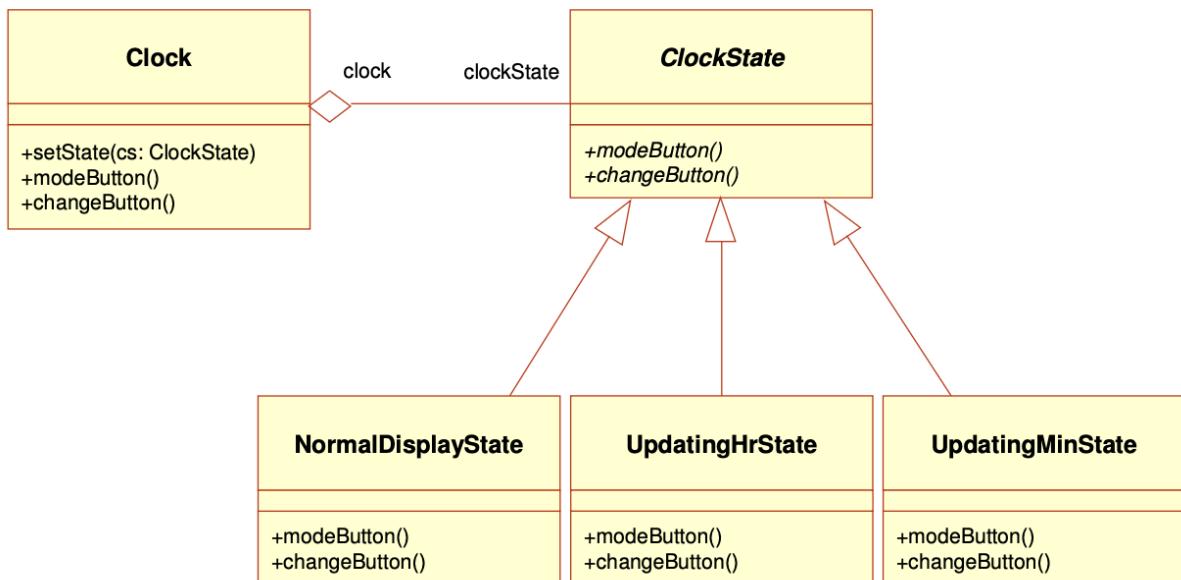
operation buttonCHANGEPRESSED{

    if( clockState = NORMAL_DISPLAY )
        displayTimeWithLight();
    else if( clockState = UPDATING_HOURS )
        hours++;
    else if( clockState = UPDATING_MINUTES )
        minutes++;
    ...
}

```

Il problema principale di questa soluzione è che diventa difficile aggiungere stati in più all'orologio. Infatti, occorre modificare il codice (aggiungere if in cascata) in modo globale. Il pattern state offre una soluzione alternativa modulare ed estensibile.

Vediamo ora un diagramma UML della soluzione tramite pattern State:



In questo esempio, la classe Context dello schema generale corrisponde alla classe Clock, la quale memorizza nel suo campo state un'istanza di tipo ClockState, rappresentante lo stato corrente. La classe astratta ClockState specifica le operazioni comuni alle classi ConcreteState, i cui oggetti encapsulano il comportamento di un oggetto Context, un comportamento diverso per ogni oggetto ConcreteState. Le

classi ConcreteState dello schema generale sono rappresentate in questo esempio dalle classi NormalDisplayState, UpdatingHrState e UpdatingMinState.

Mostramo ora un'implementazione di questo esempio in Java. La classe astratta ClockState specifica l'insieme dei metodi che ogni ConcreteState deve implementare. In particolare, essa offre due metodi modeButton() e changeButton() che rappresentano le operazioni eseguite se vengono premuti il tasto MODE o il tasto CHANGE dell'orologio. Queste operazioni hanno comportamenti diversi a seconda dello stato in cui trova l'orologio. La classe ClockState gestisce anche un riferimento "all'indietro" all'oggetto Clock (il Context) a cui appartiene, in modo che ogni stato possa accedere alle sue proprietà del Clock.

```
public abstract class ClockState {  
    protected Clock clock;  
    public ClockState(Clock clock) {  
        this.clock = clock;  
    }  
    public abstract void modeButton();  
    public abstract void changeButton();  
}
```

La NormalDisplayState estende ClockState. Il suo costruttore richiama quello della superclasse per la gestione del riferimento all'oggetto Clock. Il metodo modeButton() cambia lo stato dell'orologio da "visualizzazione normale" a "aggiornamento delle ore" (creando una istanza di UpdatingHrState e associanola allo stato dell'orologio). Il metodo changeButton() accende la luce del display (si ipotizzi che la luce si spegne automaticamente).

```
public class NormalDisplayState extends ClockState {  
    public NormalDisplayState(Clock clock) {  
        super(clock);  
        System.out.println( "*** Clock is in normal display." );  
    }  
    public void modeButton() {  
        clock.setState( new UpdatingHrState( clock ) );  
    }  
    public void changeButton() {  
        System.out.print( "LIGHT ON: " );  
    }  
}
```

```

    clock.showTime();
}
}

```

La classe UpdatingHrState rappresenta lo stato di modifica del numero delle ore dell'orologio. Il metodo modeButton() cambia lo stato a "modifica dei minuti" (creando una istanza di UpdatingMinState e associandola al Clock). Il metodo changeButton() incrementa l'ora corrente di una unità.

```

public class UpdatingHrState extends ClockState {
    public UpdatingHrState(Clock clock) {
        super( clock );
        System.out.println("## UPDATING HR: Press CHANGE button to
increase hours.");
    }
    public void modeButton() {
        clock.setState( new UpdatingMinState( clock ) );
    }
    public void changeButton() {
        clock.hr++;
        if(clock.hr == 24)
            clock.hr = 0;
        System.out.print( "CHANGE pressed -");
        clock.showTime();
    }
}

```

La classe UpdatingMinState rappresenta lo stato di "modifica dei minuti". In questo stato, il metodo modeButton() porta l'orologio allo stato di "visualizzazione normale" (tramite la creazione e associazione all'orologio Clock di una istanza di NormalDisplayState). Il metodo changeButton() incrementa di una unità i minuti dell'orologio.

```

public class UpdatingMinState extends ClockState {
    public UpdatingMinState(Clock clock) {
        super( clock );
        System.out.println("## UPDATING MIN: Press CHANGE button
to increase minutes.");
    }
    public void modeButton() {
        clock.setState( new NormalDisplayState( clock ) );
    }
}

```

```

public void changeButton() {
    clock.min++;
    if(clock.min == 60)
        clock.min = 0;
    System.out.print( "CHANGE pressed -");
    clock.showTime();
}
}

```

**La classe Clock rappresenta il Context.** Lo stato corrente di ogni istanza di Clock viene memorizzato nel campo ClockState come riferimento a un oggetto ConcreteState. Al momento della creazione, ogni clock viene settato alle ore 12:00 e in stato di visualizzazione "normale".

```

public class Clock {
    private ClockState clockState;
    public int hr, min;
    public Clock() {
        clockState = new NormalDisplayState(this);
    }
    public void setState(ClockState cs) {
        clockState = cs;
    }
    public void modeButton() {
        clockState.modeButton();
    }
    public void changeButton() {
        clockState.changeButton();
    }
    public void showTime() {
        System.out.println( "Current time is Hr : " + hr +
Min:+ min );
    }
}

```

Presentiamo ora il codice di una semplice applicazione, in cui:

1. Si preme il tasto CHANGE: dato che l'orologio è nello stato di visualizzazione normale, viene mostrata l'ora corrente con la luce del display accesa.
2. Si preme il tasto MODE: viene attivato lo stato di modifica delle ore.

3. Si Preme due volte il tasto CHANGE: cambia l'ora corrente alle ore 14.
4. Si preme il tasto MODE: si attiva lo stato di modifica dei minuti.
5. Si preme quattro volte il tasto CHANGE: cambia il numero dei minuti a 4.
6. Si preme il tasto MODE: si ritorna allo stato di visualizzazione normale.

```
public class StateExample {
    public static void main (String arg[]) {
        Clock theClock = new Clock();
        theClock.changeButton();
        theClock.modeButton();
        theClock.changeButton();
        theClock.changeButton();
        theClock.modeButton();
        theClock.changeButton();
        theClock.changeButton();
        theClock.changeButton();
        theClock.changeButton();
        theClock.modeButton();
    }
}
```

**Qualche osservazione sullo State.** In questo esempio ogni operazione crea un nuovo stato e lo assegna come stato corrente al campo `clockState`, ovvero gli oggetti che rappresentano gli stati non vengono riutilizzati, perché ogni volta che si cambia di stato viene creato un nuovo oggetto. Sarebbe più efficiente utilizzare una variante dello State in cui si memorizza una sola istanza di ogni stato concreto e si assegna quella opportuna al campo `StateClock` tutte le volte che l'orologio cambia stato. Si noti, quindi, che questa variante si appoggia al pattern Singleton che abbiamo introdotto prima: `NormalDisplayState`, `UpdatingHrState`, `UpdatingMinState` diventano tre classi Singleton.

Lo svantaggio principale del pattern State è che si aggiunge un'indirezione perché lo stato non è memorizzato direttamente nel Context, ma in un altro oggetto. Tuttavia, si guadagna molto in flessibilità, specie in presenza di molti stati concreti (compresi quelli che si vorranno aggiungere in futuro a una certa implementazione) e magari complessi.

Riportiamo l'esempio sopra avendo modificato in singleton le classi che rappresentano gli stati. Come main() potete utilizzare quello definito sopra.

```
public class NormalDisplayState extends ClockState {
    private static NormalDisplayState instance = null;

    private NormalDisplayState(Clock clock) {
        super(clock);
        System.out.println( "*** Clock is in normal display." );
    }

    public void modeButton() {
        clock.setState( UpdatingHrState.newInstance( clock ) );
    }

    public static NormalDisplayState newInstance(Clock c) {
        if (instance == null)
            instance = new NormalDisplayState(c);
        return instance;
    }

    public void changeButton() {
        System.out.print( "LIGHT ON: " );
        clock.showTime();
    }
} // fine classe NormalDisplayState

public class UpdatingHrState extends ClockState {
    private static UpdatingHrState instance = null;

    private UpdatingHrState(Clock clock) {
        super( clock );
        System.out.println("/** UPDATING HR: Press CHANGE button to
increase hours.");
    }

    public void modeButton() {
        clock.setState( UpdatingMinState.newInstance( clock ) );
    }
```

```

public static UpdatingHrState newInstance(Clock c) {
    if (instance == null)
        instance = new UpdatingHrState(c);
    return instance;
}

public void changeButton() {
    clock.hr++;
    if(clock.hr == 24)
        clock.hr = 0;
    System.out.print( "CHANGE pressed -");
    clock.showTime();
}
// fine classe UpdatingHrState

public class UpdatingMinState extends ClockState {

    private static UpdatingMinState instance = null;

    private UpdatingMinState(Clock clock) {
        super( clock );
        System.out.println("** UPDATING MIN: Press CHANGE button
to increase minutes.");
    }

    public void modeButton() {
        clock.setState( NormalDisplayState.newInstance( clock )
    };
}

public static UpdatingMinState newInstance(Clock c) {
    if (instance == null)
        instance = new UpdatingMinState(c);
    return instance;
}

public void changeButton() {
    clock.min++;
    if(clock.min == 60)
        clock.min = 0;
    System.out.print( "CHANGE pressed -");
    clock.showTime();
}

```

```

} // fine classe UpdatingMinState

public abstract class ClockState { // resta invariata
    protected Clock clock;

    public ClockState(Clock clock) {
        this.clock = clock;
    }

    public abstract void modeButton();

    public abstract void changeButton();
} // fine classe ClockState

public class Clock {

    private ClockState clockState;
    public int hr, min;

    public Clock() {
        clockState = NormalDisplayState.newInstance(this);
    }

    public void setState(ClockState cs) {
        clockState = cs;
    }

    public void modeButton() {
        clockState.modeButton();
    }

    public void changeButton() {
        clockState.changeButton();
    }

    public void showTime() {
        System.out.println( "Current time is Hr : " + hr + "
Min:" + min );
    }
} // fine classe Clock

```

**Avevamo già incontrato un pattern.** Nella Lezione 15 e nella Lezione 20 abbiamo costruito incrementalmente due versioni semplificate ma fedeli di una struttura-dati che fa parte delle Collections di Java. Infatti, MiniLinkedList (con MiniIterator) e ListExt (con ListIterator) non sono altro che due versioni "didattiche" di **LinkedList** (<https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/LinkedList.html>) .

Un oggetto della classe `LinkedList`, che implementa l'interfaccia `Iterable<E>`, può essere attraversata elemento per elemento **solo** tramite l'uso di un oggetto iteratore di tipo interfaccia `ListIterator<E>` (che a sua volta estende l'interfaccia `Iterator<E>`).

Questa accoppiata `Iterable-Iterator` non è altro che l'implementazione Java di un altro pattern GoF, il **pattern Iterator**.

# **Lezione 23**

## **Esercizi**

 Esercizi d'esame

# Appendice

## (NON FA PARTE DEL PROGRAMMA DEL CORSO)

**Cenni ai concetti di covarianza,  
controvarianza, invarianza. Introduzione alle  
wildcard**

Se volete approfondire l'argomento del subtyping, potete dare un'occhiata ai seguenti appunti:

 [Note\\_sul\\_subtyping.pdf](#)

Per avere un'introduzione sulle wildcard, potete utilizzare questa risorsa:

[Wildcards - The Java™ Tutorials](#)

Infine, ecco un esempio che mette insieme il subtyping tra tipi funzionali, introdotto nelle Note sul subtyping, e le wildcard:

[Prova.java](#)