

Progettazione Del Software

Simone Palumbo

September 2021

Contents

I Introduzione alla progettazione del Software:	7
1 Contesto Organizzativo:	7
1.1 Quali sono gli attori della progettazione del SW?	7
1.2 Classificazione delle applicazioni:	7
2 Ciclo di vita del Software:	8
3 Qualità del Software:	9
3.1 Principi Guida nello Sviluppo del SW:	10
4 Modularizzazione:	11
4.1 Modulo:	11
4.2 Principi alla base della modularità:	11
4.3 Gli effetti di una buona modularizzazione:	14
5 Principi di base dell'orientazione ad oggetti:	14
5.1 Parole chiave dell'approccio OO:	14
5.2 Fattori di qualità del Software influenzati dall'approccio OO:	15
6 Esercizi di fine capitolo:	16
II Introduzione a Java come Linguaggio OO:	17
1 Programma e Paradigmi di Programmazione:	17
2 Particolarità di Java:	17
3 Scrittura, Compilazione ed Esecuzione di Programmi in Java:	17
3.1 Preparazione del Testo del Programma:	17
3.2 Compilazione del Programma, generamento del ByteCode:	18
3.3 Esecuzione del programma, esecuzione del ByteCode:	18
3.4 Facciamo un esempio:	18
4 IDE:	20
4.1 Esempio con l'IDE Eclipse:	20
4.2 Esempio con l'IDE NetBeans:	22
5 API Java:	23
6 Dispiegare programmi in Java:	23
III Le Classi in Java:	24
1 Che cos'è una classe:	24

2 Come si usa una classe?	24
2.1 Riferimenti e Allocazione:	25
3 Campi Public e Private:	26
4 Overloading:	28
5 Costruttori:	29
6 Inizializzazioni Implicite dei campi dati di una classe:	30
7 Esercizio:	31
7.1 Esercizio, Client con due Metodi Static:	32
8 Il riferimento This:	32
9 Esercizi:	33
10 Package:	34
10.1 Sottopackage:	35
11 Livelli di accesso:	35
 IV Ereditarietà:	37
12 Derivazione tra Classi:	37
12.1 Principi Fondamentali della Derivazione:	37
12.2 Gerarchie di classi:	38
12.3 Casting:	38
12.4 Livello di Accesso Protected:	39
12.5 Costruttori delle Classi Derivate:	39
12.6 Costrutto this():	41
13 Overriding:	41
14 Late Binding:	43
14.1 L'overriding di campi dati non esiste!	45
15 Classi Astratte:	46
15.1 Cliente di Classe Astratta e Polimorfismo:	49
 V Polimorfismo:	51
16 Interfacce Java:	51
17 La Classe Object:	55
18 La Classe Class:	57
19 La classe String Tokenizer, le matrici e il ciclo for each:	58
20 Uguaglianza tra Oggetti:	60
20.1 Metodo equals():	61
20.2 hashCode e overriding di hashCode:	62
20.3 Overriding di equals() nelle classi derivate:	63
21 Copia di Oggetti:	64
21.1 Overriding di clone() nelle classi derivate:	65

22 Oggetti Mutabili e Immutabili:	66
22.1 Oggetti mutabili:	66
22.2 Oggetti immutabili:	66
22.3 Equals() e Clone() per oggetti Mutabili e Immutabili:	66
22.3.1 Oggetti Mutabili:	66
22.3.2 Oggetti Immutabili:	66
23 Astrazione di Valore ed Astrazione di Entità in Java:	67
23.1 Astrazione di Valore:	67
23.1.1 Astrazione di Valore (Semplice):	67
23.1.2 Astrazione di Valore Complesso (collezioni):	68
23.1.3 Astrazione di Entità:	68
23.2 Schemi Realizzativi:	69
23.2.1 Il caso dell'interferenza:	69
23.2.2 Sharing:	70
23.3 Schemi realizzativi di valore (semplice) ed entità (e valori complessi), quando usare Side e Condivisione?	71
23.4 Equals() e Clone() in questi quattro schemi realizzativi:	71
23.5 Implementazioni FC, SS, SC di Insieme:	72
23.5.1 Implementazione FC:	72
23.5.2 Implementazione SS:	83
23.5.3 Implementazione SC???	87
23.6 Implementazione di tipo di dato astratto studente:	87
23.6.1 Postilla: come avevo fatto io inizialmente questo codice:	90
24 Gestione delle Eccezioni in Java:	95
24.1 Gerarchia di Classi in Java:	95
24.2 Eccezioni Controllate (checked) e Non Controllate (Unchecked):	95
24.3 Ok ma, come gestire operativamente le eccezioni?	96
24.3.1 Dettagli sulla classe Exception:	97
24.4 Un bell'esempio conclusivo:	97
24.5 Progettare nuove eccezioni:	101
24.6 Catturare eccezioni:	102
24.6.1 La clausola Finally:	102
VI Java Collections Framework:	103
25 Tipi Generici (Generics):	103
25.1 Tipi Generici e Sottoclassi:	105
25.2 Cancellazione (Erasure):	106
26 Java Collections Framework:	106
26.1 Interfacce del JCF:	108
26.1.1 Interfaccia Collection:	108
26.1.2 Iteratori:	110
26.1.3 Interfaccia Iterable:	112
26.2 Mappe:	112
26.3 Collezioni Ordinate:	113
26.4 Strutture Dati per l'implementazioni delle interfacce:	114
26.5 Domande su Wildcard e modifica collezione durante scansione con iteratore:	115
27 Figura Geometrica, Completa:	116
28 Esercizio Banca:	123
28.1 Modifica esercizio Banca con JCF:	129
29 Laboratorio 6: Implementazione SS di MioSet e MiaLista	131
VII Interfacce Grafiche:	145

30 JFC e Swing:	145
30.1 Java Swing:	145
31 La Classe JFrame:	146
31.1 Struttura a livelli di un oggetto JFrame:	147
31.2 Popolare un JFrame:	148
31.3 Modularita' delle GUI:	149
32 Layout Managers:	150
32.1 Classe FlowLayout:	150
32.2 Classe GridLayout:	151
32.3 Classe BorderLayout:	152
33 Progettazione della GUI:	154
33.1 Container Annidati e JPanel:	154
33.2 Alcuni JComponent Comuni:	157
33.2.1 JTextComponent (JComponent testuali):	157
34 I Menu: Progettazione e Realizzazione:	158
35 Eventi e Ascoltatori:	161
35.1 Event Delegation:	161
35.2 Facciamo un esempio con il Tipo di eventi MouseEvent:	162
35.3 Limitare la proliferazione dei Listener:	171
36 Laboratorio 7 dell'11-11:	174
36.1 Esercizio 1:	174
36.2 Esercizio 2:	179
36.3 Shortcut da Tastiera:	182
37 L'istruzione Switch:	182
VIII Logging:	184
37.1 Concetti Principali del Logging:	184
37.1.1 Messaggio di Logging (log records):	184
37.1.2 Logger:	184
37.1.3 Hadler (o Appender):	184
37.1.4 Livello di logging (filter):	184
37.2 Il Sistema di Logging Standard di Java: java.util.logging:	185
37.2.1 La classe LogManager:	185
37.3 Configurazione del Logging:	185
37.4 Vediamo un esempio di Programma che fa Logging:	186
37.5 Un altro esempio:	186
37.6 Personalizzare il logging:	187
37.7 Impostare un livello di gravità per un logger:	192
IX Input/Output:	193
37.7.1 Il problema dell'Endianness (Ordine dei Byte):	193
37.8 Stream di dati e di manipolazione:	193
37.9 La Gerarchia delle Classi Stream:	194
37.10InputStream:	195
37.11OutputStream:	195
38 Stream di Caratteri:	195
38.1 Reader:	196
38.2 Writer:	196
39 I/O Standard:	196

40 File:	198
40.1 Lettura di un File di Testo, Esempio:	199
40.2 StringTokenizer:	199
40.3 Scrittura di un file di Testo, esempio:	202
X Sockets e rudimenti di programmazione di rete:	203
41 Rete di Calcolatori:	203
41.1 Paradigmi di Comunicazione:	203
41.2 Identificazione dei nodi:	203
41.3 Servizi e Sever:	204
42 Socket:	204
42.1 Esercizio, Echo Server:	205
42.2 Esercizio Lotto:	207
42.3 Classe InetAddress:	218
42.3.1 Chiudere le Connessioni:	218
43 Thread, concetti di base:	219
43.1 Condivisione di Memoria nei thread:	219
43.2 Thread in Java:	219
43.3 Implementazione dei Thread in Java, i due modi:	220
43.4 Esercizio:	222
43.5 Variabili nei Thread:	223
43.6 Ciclo di Vita del Thread:	224
XI Seconda Parte, l'analisi:	226
44 Diagramma delle attività:	227
45 Diagramma degli Stati e delle Transizioni:	227
46 Diagrammi delle classi e degli oggetti:	227
46.1 Rappresentazione di Classi e Oggetti in UML:	227
46.1.1 Gli oggetti:	227
46.1.2 Le classi, gli attributi e Instance-Of:	227
46.1.3 Facciamo un Esempio:	230
46.1.4 Altro esempio, differenza tra attributi e associazioni:	230
46.2 Versi e doppi nomi nelle Associazioni:	231
46.3 Ruoli nelle associazioni:	232
46.4 Attributi di Associazione:	232
46.5 Molteplicità delle associazioni:	233
46.6 Molteplicità degli attributi:	234
46.7 Esercizio: Dipartimenti Aziendali	234
46.8 Associazioni n-arie:	235
46.8.1 Esempio:	235
46.9 Associazioni Ordinate:	238
46.9.1 Facciamo un esempio:	239
46.10 Generalizzazioni:	240
46.10.1 Ereditarietà:	240
46.10.2 Generalizzazione tra più di due classi:	240
46.10.3 Diverse generalizzazioni della stessa classe:	240
46.11 Istanze Comuni ed Ereditarietà Multipla:	242
46.11.1 Facciamo un esempio:	242
46.12 Differenza tra due is-a e una generalizzazione:	243
46.13 Specializzazione:	243
46.13.1 Quando si eredita il vincolo di molteplicità minimo nella Specializzazione di Associazioni?	244
46.14 Operazioni:	245

46.14.1 Facciamo un esempio:	245
46.14.2 Specializzazione di Operazioni:	246
47 Tipi più complessi:	246
48 Riassunto: tutti gli elementi del Diagramma delle Classi	247
49 Metodologia per la Costruzione del Diagramma delle Classi:	247
49.1 Controllo di Qualità:	247
XII Fase di Progetto:	249
49.2 Input della fase di progetto:	249
49.3 Output della fase di progetto:	250
49.4 La responsabilità:	250
49.5 Scelta e Progettazione delle Strutture Dati:	250
49.6 Tipi:	250
49.7 Gestione delle proprietà:	250
49.8 API:	250
49.9 Studio di caso: scuola elementare	251
49.9.1 Vediamo le responsabilità in questo caso:	252
49.10 Responsabilità sui Ruoli:	253
50 Strutture Dati:	254
50.1 Come trovare se e dove servono:	254
50.2 Come farle:	254
51 I Tipi, da UML a Java:	254

Part I

Introduzione alla progettazione del Software:

1 Contesto Organizzativo:

1.1 Quali sono gli attori della progettazione del SW?

Molte volte nelle imprese tutte o solo alcune di queste figure coincidono con una persona sola.

- Committente: chi commissiona lo sviluppo. Ad esempio un'azienda esterna, un'istituzione ecc.
- Esperti del dominio: sono coloro che forniscono a chi deve progettare il SW i requisiti. Ad esempio se bisogna progettare un'applicazione per un ospedale - tenendo ovviamente conto che il team che progetterà l'app è formato da esperti di informatica e non di medicina - gli esperti del dominio saranno dei medici o altri professionisti che, essendo esperti di medicina, sanno come l'app dovrebbe reagire a un determinato input.
- Analista: prende i requisiti degli esperti del dominio, li analizza e li mette in una forma ordinata, *ma solo testuale! Non tecnica*, a quella ci pensa il progettista.
- Progettista: prende il progetto dell'analista e lo completa con il formalismo tecnico (ad esempio usando grafici).
- Programmatori: qui si inizia a scrivere il codice.
- Utente finale: il prodotto è pronto per essere usato da questi
- Manutentore: è un programmatore che risolve i bachi¹ che si presentano durante l'utilizzo.

1.2 Classificazione delle applicazioni:

Tenendo in considerazione che non si tratta di camere stagne, ma molto spesso questi aspetti convivono in una singola applicazione.

- Classificazione rispetto al flusso di controllo:
 - Applicazioni Sequenziali: un unico flusso di controllo governa l'evoluzione dell'applicazione. Cioè l'applicazione parte e finisce con una *task*, un processo. In generale tutto ciò che è matematica e analisi di dati è sequenziale
 - Applicazioni Concorrenti: più processi allo stesso tempo, cioè più attività sequenziali contemporaneamente. Le varie attività sequenziali devono essere tra loro sincronizzate e devono poter comunicare.
 - Applicazioni Dipendenti dal Tempo: si rifanno a sistemi sensibili (es. gestione centrali nucleari) e devono perciò rispondere in tempo reale.
- Classificazione rispetto agli elementi di interesse primario:
 - Applicazioni orientate alla realizzazione di funzioni: la complessità è sulle funzioni da realizzare, che poi saranno usate dall'app.
 - Applicazioni orientate alla gestione dei dati: il focus dell'applicazione è quello di memorizzare, ricercare e modificare i dati.
 - Applicazioni orientate al controllo: la complessità ricade principalmente sul controllo delle attività sequenziali che si sincronizzano e cooperano.

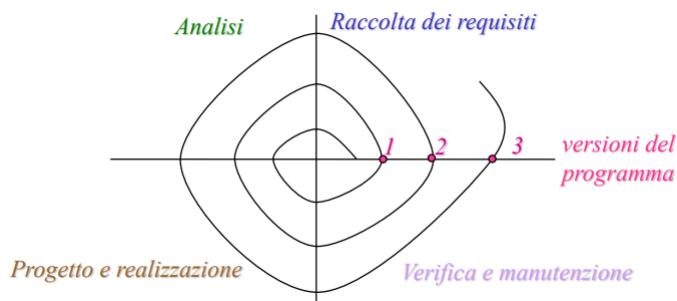
In questo corso ci concentriamo sulle applicazioni sequenziali e concorrenti (per quanto riguarda la prima classificazione) e orientate alla realizzazione di funzioni (per la seconda classificazione). Queste sono le applicazioni più comuni.

¹Bug in italiano.

2 Ciclo di vita del Software:

la vita di un SW è scandita da queste fasi:

- Fattibilità e raccolta dei requisiti: in questa fase, dopo che un eventuale committente ha commissionato l'opera
 - si valutano i costi e i benefici
 - si raccolgono i requisiti dagli esperti del dominio
 - si pianificano le attività e le risorse per il progetto
- Analisi Requisiti: in questa fase bisogna chiarire il marasma di informazioni e requisiti raccolti durante la fase precedente. Questa è la fase in cui l'analista spiega **cosa**, cioè quali specifiche, l'applicazione dovrà realizzare.
- Progetto e realizzazione: in questa fase ci si occupa del **come** l'applicazione dovrà realizzare il cosa. Quindi si fanno schemi precisi su come fare il cosa e si progetta l'architettura del programma. Dopo questa progettazione inizia la vera e propria programmazione, in cui si scrive il codice del programma e si produce la documentazione.
- Verifica: si prende il SW programmato e lo si testa. Il programma svolge correttamente, completamente e efficientemente il compito? Se sì vai avanti, se no *torna alla fase di raccolta dei requisiti*. Questo è il cosiddetto **modello a spirale**², dove, dopo una verifica fallimentare, *si ritorna alla raccolta dei requisiti*, proponendo per ogni volta un SW sempre migliore e raffinato.



- Manutenzione: una volta che svolge il suo esercizio, il programma viene costantemente controllato, corretto e aggiornato, fino alla sua morte. Questo perché gli utenti scoprono sempre nuovi bachi.

²Questo modello a spirale oggi è vetusto in realtà, e nel mercato lo sviluppo viene fatto a sprint: fai subito un applicazione e la metti nel mercato, poi mentre è sul mercato la continui ad aggiornare e migliorare. Questo è il modello *agile*.

3 Qualità del Software:

la linea tra qualità interne e esterne non è nettamente marcata, poiché se la qualità interna è pessima, questo influenza negativamente anche la qualità esterna, visibile dagli utenti.

Q. Esterne, viste direttamente dall'utente:

- Correttezza: il software fa quello per cui è stato progettato
- Affidabilità: si può fare affidamento su ciò che dice il SW. Se si parla di una calcolatrice l'affidabilità è la capacità di dare il risultato corretto.
- Robustezza: si comporta bene anche nel caso di situazioni non previste
- Sicurezza: web security, quindi riservatezza nell'accesso alle informazioni
- Innocuità: sicurezza fisica.
- Usabilità: il focus su utente e sua psicologia. Si cerca di capire la sua mentalità, e in quale situazione lui ha più comodità. Cura delle interfacce grafiche.
- Estendibilità: facilità con cui posso aggiungere al SW nuove funzionalità, letteralmente estenderlo.
- Riusabilità: facilità con cui posso reimpiegare il SW in applicazioni diverse da quella originaria. Quindi un SW compatibile con macchine ad esempio. Ho delle parti di codice posso riutilizzarle in un altro applicativo. Ho un modulo complesso che posso riutilizzare in un altro modulo.
- Interoperabilità: la facilità di ogni modulo del SW di comunicare e sincronizzarsi con gli altri moduli per svolgere un compito insieme.

Nota che correttezza, estendibilità e riusabilità (per le qualità esterne); strutturazione e modularità (per le qualità interne) sono le *caratteristiche fondamentali in un SW*, e sono estremamente favorite dall'approccio orientato agli oggetti che sviluppiamo con Java.

Q. interne, serve ispezione del codice:

- Efficienza: quanto pesa, quanto costa il SW in termini di tempo di esecuzione e utilizzo di memoria (cioè le risorse di sistema). Per studiare la complessità di un programma c'è la Teoria della complessità (vista già in parte nei corsi di Fondamenti di Informatica I e Tecniche di Programmazione, ma la vedremo per bene nel corso di Fondamenti di Informatica II). Ovviamente il tempo e la memoria utilizzata da un programma influenzano le prestazioni dell'app, che vengono percepite dall'utente.
- Strutturazione: l'architettura del SW è strutturata in macrocontenuti.
- Modularità: Ogni singolo macrocontenuto è un modulo, e i moduli tra loro comunicano, per svolgere un compito complesso insieme. Quindiciamo che la strutturazione mette il focus sul fatto che il SW è fatto di macrocontenuti, mentre la modularità specifica che questi macrocontenuti sono moduli, e che comunicano tra loro.
- Comprensibilità: capacità del SW di essere compreso anche da chi non ha condotto il progetto. Un SW è sicuramente più comprensibile se ha strutturabilità e modularità
- Verificabilità: possibilità e facilità nell'eseguire la fase di verifica del SW.
- Manutenibilità: possibilità e facilità nell'eseguire la fase di manutenzione del SW.
- Portabilità: facilità nell'operare su diverse piattaforme. Un applicazione scritta su Windows che gira anche su Linux è portatile. Java favorisce la portabilità. Quindi portabilità è operare su diverse piattaforme, riusabilità è poter usare moduli di codice per altri moduli del SW o per un altro SW.

La **misura delle qualità** deve essere valutata secondo **oggettivamente** e **quantitativamente**, ad esempio un parametro per valutare la comprensibilità è la percentuale di commenti nel codice.

Inoltre notare che non tutte le qualità possono essere massimizzate, poiché alcune sono in **contrasto** tra di loro. Per esempio usabilità e sicurezza³, o efficienza e portabilità⁴. Per questo motivo è necessario bilanciare le qualità del SW. Le qualità dei programmi posso anche aiutarsi tra loro, ad esempio nel caso della modularità che aumenta la comprensibilità.

Tendenza nello sviluppo di applicazioni SW: oggigiorno si tende a:

- + Complessità: le informazioni da processare diventano più complesse
- + Progetti di medio-grandi dimensioni: i progetti di piccole dimensioni diventano sempre di più i singoli componenti dei progetti che poi vengono rilasciati
- + Eterogeneità degli utenti: oggi usufruiscono di una stessa applicazione utenti dei più diversi livelli di esperienza. Quindi ti trovi lo smanettone e il novizio.
- - Durata media dei sistemi: come dicevamo, il tempo di vita del SW sta sempre più diminuendo
- + Manutenzione: essendoci più utenti e anche più eterogeneità di questi, essi "scoprono" più bug e più facilmente, quindi c'è bisogno di più manutenzione
- - Costi di produzione e tempi di produzione: i tempi si abbassano anche grazie all'utilizzo di metodologie AGILE, favorite rispetto al modello a spirale.
- + Qualità del prodotto finale: anche grazie all'aumento degli interventi di manutenzione

3.1 Principi Guida nello Sviluppo del SW:

Terminiamo la sezione dedicata alla qualità del SW parlando, in base a tutte le caratteristiche descritte sopra, di quali sono gli aspetti che oggigiorno si considerano fondamentali nello sviluppo del SW. Questi principi dovrebbero guidare lo sviluppo: dovrebbero perché nel mondo del lavoro molte cose si fanno all'acqua di rose senza seguire delle metodologie di progettazione.

- Rigore e Formalità: il rigore tecnico e la formalità logico-matematica permettono di creare prodotti affidabili. Il sogno dell'informatica è quello di non programmare più gli applicativi ma progettare dei modelli formali e poi farli tradurre in codice da un compilatore. Microsoft ha pubblicato un progetto: hanno preso tutto il codice di github e messo dentro una rete neurale, poi voi scrivete il nome di una funzione e la rete scrive tutto il codice. Il progetto forse lo mette nei link interessanti del sito del corso. Quest'attenzione al rigore e alla formalità è predominante nella fase di progetto, mentre nella fase di analisi si resta discorsivi (l'analista ricordiamo che prende i requisiti e li mette in una forma testuale decente più ordinata del marasma che fanno gli esperti del dominio).
- Separazione degli Interessi (*separation of concerns*): per dominare e gestire la complessità di un SW bisogna affrontare separatamente i diversi aspetti di un SW. Lo sviluppo consta in effetti di due macrofasi: prima sviluppiamo i blocchetti (cioè i moduli) che separano gli interessi, e poi nella seconda macrofase si creano i blocchetti (sempre altri moduli) che servono ad orchestrare i moduli della prima sezione tra di loro, per farli cooperare. Collegati a questo ci sono ovviamente i concetti di strutturazione e modularità
- Astrazione: identifica gli aspetti fondamentali e tralascia dettagli irrilevanti. Nella fase di progetto si intende: cioè si ragiona per classi (dopo vedremo questo concetto), definendo degli insiemi che delineano solo le proprietà fondamentali dei loro oggetti, e non i singoli dettagli di essi. Ovviamente l'attenzione ai dettagli invece c'è nella fase di programmazione.
- Anticipazione del cambiamento: si intende il cambiamento del mondo esterno. Cioè è tendenza a immaginare già possibili sviluppi del SW mentre lo sto sviluppando. Es. se sto sviluppando SW per gestione multe, ma so che c'è una nuova legge per le multe in parlamento, già mi preparo per quando dovrò passare da una legge all'altra, non è che inizio a lavorarci nel momento in cui è entrata in vigore la legge.

³Prendiamo l'esempio dell'autenticazione a due fattori, con cui rendiamo più sicura l'app, ma più scomoda e lenta per l'utente, e quindi meno usabile.

⁴Più portabilità significa operare ad alto livello, e più si opera ad alto livello più e lento, e quindi meno efficiente, il codice. Al contrario, un codice che comunica ad un livello più vicino alla macchina, magari proprio a livello macchina, è più efficiente, ma meno portatile perché più legato appunto alla particolare macchina su cui è stato scritto

- Generalità: si ricercano soluzioni generali, in modo da essere portabili. Ovviamente è collegato al concetto di anticipazione del cambiamento e a quello di riusabilità: se faccio soluzioni generali mi posso rigiocare la soluzione per quando dovrò estendere il SW.
- Incrementalità: oggigiorno lo sviluppo SW è incrementale, cioè si cerca di implementare poche funzionalità, poi verificarle, poi implementarne un altro po', verificare queste e così via, fino a prodotto finito, piuttosto che implementare tante funzionalità e poi verificarle tutte assieme, con una probabilità di averle implementate bene più bassa. Diciamo che le metodologie AGILE ben si addicono all'incrementalità: implemento queste 3 funzioni, le verifico, funzionano, pubblico; nel mentre che è pubblicato il SW implemento altre 3, le verifico, aggiorno, e così via.

4 Modularizzazione:

Iniziamo questa sezione dicendo che *modulo* è un termine abusato in informatica, spesso confuso con componente. Sono cose simili, ma hanno sfumature diverse: un modulo è un pezzo di SW che compilo insieme al SW, il componente ha vita propria ed è disaccoppiato dal SW, è più simile ad un plug-in. Quindi gira dentro il SW ma non è compilato insieme al SW.

4.1 Modulo:

Un modulo è un'unità di un programma che:

- Fa una specifica, determinata, cosa
- Ha relazioni con altri moduli, poiché si comporta da *server* ma anche da *client*. I concetti di Client e Server, sono più generale di quelli di Client e Server del SW di rete. Anche nella singola applicazione, non connessa a internet, ho sia client che server

Ricordiamo invece che in C un modulo era un file, es. *funzionimatematiche.c*, un file che contiene delle funzioni matematiche, che poi possiamo usare in un programma, era un modulo. Per gli obiettivi di questo corso invece, *il modulo è una classe di Java*. C'è da dire che noi nel corso vedremo che la classe è un file, quindi sembra che modulo in C e in Java siano la stessa cosa, ma in generale questo non è vero: cioè non sempre una classe in Java è un file.

Abbiamo già detto che spesso si fa confusione tra componente di un SW e modulo di un SW; complichiamo ancora di più: esistono anche le componenti di un modulo :). Per componenti di un modulo si intendono le parti da cui è composto. Vediamo una figura esplicativa: Questo tipo di figure è molto ricorrente per la rappresentazione dei moduli in informatica, quindi nella nostra carriera la rivedremo sicuro.

4.2 Principi alla base della modularità:

Vedremo che sono gli stessi principi della programmazione orientata a oggetti.

- Unitarietà: deve incorporare tutti e soli gli aspetti del modulo
- Numero giusto di interfacce: non bisogna strafare con la modularizzazione
- Poca comunicazione tra moduli: o meglio, la quantità necessaria e non più di comunicazione tra moduli. Questa proprietà, insieme alla proprietà del numero giusto di interfacce, forma il *basso accoppiamento*.
- Comunicazione chiara: interfacce chiare e ben documentate. Es. le interfacce devono avere dei nomi facilmente comprensibili, anche guardando solo segnatura deve essere chiaro ed esplicito di cosa l'interfaccia parla.
- Information hiding: cose che non sono essenziali alla comunicazione non devono essere comunicate da un modulo all'altro.

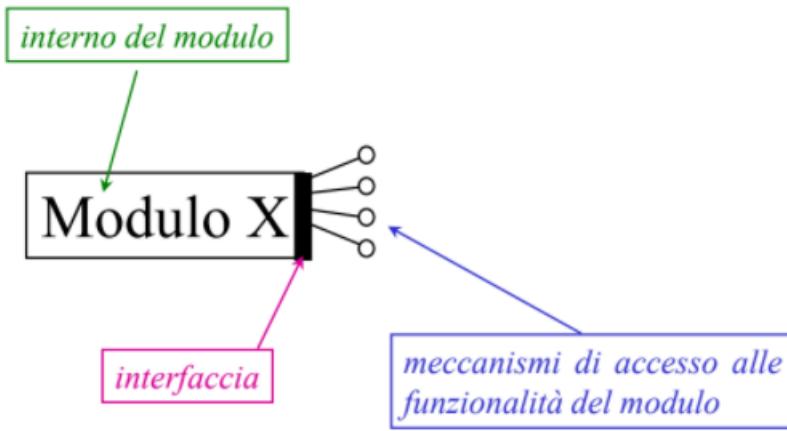


Figure 1: La rappresentazione consta di un blocco rettangolare per il nome e poi una o più strisce, ognuna indicante un’interfaccia (API in inglese, cioè *application programming interface*). Su queste interfacce c’è un insieme di cosiddetti *lollipop*, che rappresentano le varie operazioni che posso fare, cioè le varie funzionalità della mia interfaccia.

Quindi, in sintesi, questi sono i 4 dogmi. Ora vediamo degli esempi, non per forza in scenari SW, ma magari di roba quotidiana, quindi da prendere come un’analogia di ciò che succede nello sviluppo SW:

- Esempi negativi:

- Esempio di bassa coesione: ho un modulo, quello dello sportello dell’ufficio postale, che non è omogeneo, perché implementa funzionalità che non c’entrano nulla tra loro. Noi vogliamo alta coesione
- Esempio di alto accoppiamento: tabaccaio. Si tratta di un qualcosa di negativo ma spesso intrinsecamente necessario, come in questo caso.

- Esempi positivi:

- interfacciamento esplicito: qui si parla di comunicazione tra SW e utente finale, noi parlavamo della comunicazione tra moduli software all’interno dell’SW. Però è un esempio per far capire
- information hiding: voi non sapete cosa c’è scritto precisamente nella banda magnetica, ma perché non vi interessa.

Ora, finalmente, iniziamo a vedere del codice Java, e lo facciamo continuando a fare esempi di buona e cattiva modularizzazione. Java nasce come frammento del C quindi in questi esempi vedremo come la sintassi e le parole chiave di Java siano simili a quelle di C.

Esempio di cattiva modularizzazione:

Lato Server:

```

public class Server{
    public static int alfa;
    public static bool Cerca(Lista x){
        do { if (x.info == alfa) return true;
              x = x.next;
        }
        while (x != null);
        return false;
    }
}

```

```
}
```

Lato Client:

```
public class Client{
    private static Lista leggiListaCollegataInteri() {...}
    private static int leggiIntero() {...}

    public static void main(String[] args) {
        Lista s = leggiCollegataInteri();
        if (s != null) {
            Server.alfa = leggiIntero();
            if (Server.Cerca(s))
                System.out.print(Server.alfa+" presente");
            else
                System.out.print(Server.alfa+" non presente");
        }
    }
}
```

Il modulo client legge lista di interi e li passa al server, che deve restituire vero o falso in base a se c'è il valore alpha dentro la lista. Alpha è memorizzato nel modulo, cioè nella classe server. Non te fa troppe domande per ora anche perché ok che la sintassi è simile ma molte cose ancora non lei sai, ad esempio lo static in java è un'altra cosa, diversa da quella che hai studiato in C. Caratteristiche di questo codice sono

- C'è alto accoppiamento (negativo, noi vogliamo basso accoppiamento) perché Non ci va bene l'alto accoppiamento perché significa che devo fare più chiamate e quindi peggioro le performance. chiamata, chiamare cerca e passargli il valore.
- C'è basso interfacciamento esplicito (negativo, noi vogliamo alta esplicità) perché Spieghiamo perché questo è negativo con un esempio: se io sono un programmatore, e mi danno un lavoro da fare dicendomi "guarda il server - che è stato implementato da un'altra società o almeno da un'altra persona - ha questi metodi", se questi non sono chiari, io non riesco a capirli.
- C'è bassa coesione: il server non sta facendo tutto quello che gli compete, non sta controllando se la lista è nulla. Il prof ha detto che questa cosa è borderline, nel senso che non è proprio rientrante al 100% nella bassa coesione.

Esempio di buona modularizzazione:

Lato Server:

```
public class Server{
    public static bool Cerca(Lista x, int alfa){
        while (x!=null)
        { if (x.info == alfa) return true;
          else x = x.next;
        }
        return false;
    }
}
```

Lato Client:

```
public class Client{
```

```

private static Lista leggiListaCollegataInteri() {...}
private static int leggiIntero() {...}

public static void main(String[] args) {
    Lista s = leggiCollegataInteri();
    System.out.print("Inserisci un intero: ");
    int alfa = leggiIntero();
    if (Server.Cerca(s, alfa))
        System.out.print(alfa+"presente");
    else
        System.out.print(alfa+"non presente");
}
}

```

Vediamo che i difetti sono stati migliorati:

- La funzione cerca diventa più chiara quindi alto interfacciamento esplicito
- Bassa coesione e alta coesione dice il prof che è borderline come cosa, ma al limite va bene.
- L'interazione tra i due moduli è più pulita.

Tutti e due gli esempi fanno la stessa cosa, ma la qualità non è la stessa.

Static: Vediamo che nei codici sovrastanti è presente la parola chiave static. Che cos'è un metodo static, o una variabile static, in Java? Una variabile o un metodo static possono essere chiamati direttamente sulla classe e non c'è bisogno di creare un'istanza, un oggetto di quella classe.

4.3 Gli effetti di una buona modularizzazione:

- Distribuzione: tu sviluppi questo e questo, tu questo e questo, alla fine ci sarà qualcuno che si occuperà di integrarli, cioè di orchestrarli tra loro.
- Rilevare e correggere errori è più semplice.
- Negatività: un SW modulare è meno efficiente di uno non modulare: perché si porta dietro le chiamate di una funzione, quindi un costo. I vantaggi però sono imparagonabili con questo.

5 Principi di base dell'orientazione ad oggetti:

Parliamo ora dei principi di base dell'OOP (*Object Oriented Programming*). Noi per ora siamo abituati a programmare con l'approccio orientato alle funzioni: un sistema è costruito partendo dalle funzioni che manipolano dati. Nella OOP invece il rapporto è invertito e il focus è sugli oggetti: prima definiamo gli oggetti da manipolare e poi i metodi (le funzioni vengono chiamate così) che li manipolano. Quindi nell'approccio OO non ci chiediamo cosa fa il sistema ma a cosa serve, di quali oggetti è composto e come si opera su essi. Per questa sua caratteristica esso si sposa bene con la gestione di informazioni. I principi di base dell'approccio OO sono gli stessi della modularizzazione.

5.1 Parole chiave dell'approccio OO:

- Oggetto: sono dei dati sui quali si possono effettuare operazioni. Un loro sinonimo è istanza.
- Classi: gruppi di oggetti. Ne descrivono solo l'essenziale dominio di interesse. La classe è il concetto unificante di tutti i suoi oggetti, poiché ne definisce le caratteristiche strutturali (attributi) e comportamentali (metodi e operazioni).
- Astrazione: meccanismo con cui associano le caratteristiche comuni di un oggetto assegnandolo ad una classe
- Ereditarietà: le classi sono in una gerarchia. Esempio: osservare avere la classe studente e la classe persona. La classe degli studenti, essendo persone, erediterà tutte le caratteristiche della classe persona. Inoltre le caratteristiche ereditate possono essere specializzate per i figli.

- Polimorfismo: possibilità di eseguire funzioni con lo stesso nome ma specializzate per una particolare classe. Esempio: se ho la classe Persona e la classe Studente, e definisco il metodo esperienza per la classe Persona, posso implementarlo specializzato per la classe studente. Altro esempio: metodo calcolo area, ho classe forma geometrica, figli di questa classe sono triangolo, quadrato ecc.. all'interno di ogni classe ridefinisco il metodo calcolo area in base all'istanza specifica. Il vantaggio del polimorfismo è che posso iterare su queste forme geometriche con il metodo e questo si traduce nell'esecuzione corretta per ogni specifica classe. Quindi uso un solo metodo e poi fa lui

5.2 Fattori di qualità del Software influenzati dall'approccio OO:

Qualità esterne:

- Estendibilità: viene favorita dall'ereditarietà.
- Riusabilità: favorita dal concetto di classe ed ereditarietà.

Qualità interne:

- Strutturazione: favorita dal concetto di encapsulamento (codice in una classe, cioè in un modulo) e ereditarietà.
- Modularità: favorita dal concetto di classe.
- Comprensibilità: favorita perché la divisione in classi astratte è il modo in cui vediamo la realtà.

6 Esercizi di fine capitolo:

Esercizio 1: Il Comune di XYZ intende automatizzare la gestione delle informazioni relative alle contravvenzioni elevate sul suo territorio. In particolare, intende dotare ogni vigile di un dispositivo palmare che gli consenta di comunicare al sistema informatico il veicolo a cui è stata comminata la contravvenzione, il luogo in cui è stata elevata e la natura dell'infrazione. Il sistema informatico provvederà a notificare, tramite posta ordinaria, la contravvenzione al cittadino interessato. Il Comune bandisce una gara per la realizzazione e manutenzione del sistema, che viene vinta dalla Ditta ABC. Quali sono gli attori coinvolti in questa applicazione SW?

Esercizio 2: Considerare i seguenti contesti applicativi:

- sistema di controllo di una centrale nucleare
- sistema di prenotazione dei voli di un aeroporto
- risolutore di sistemi di equazioni
- sistema di gestione di una banca dati
- sistema operativo di un elaboratore elettronico

Per ciascuno di essi, fareste ricorso ad applicazioni sequenziali, concorrenti o dipendenti dal tempo?

Esercizio 3: Considerare le seguenti proprietà di un impianto HiFi:

1. Tempo dedicato al collaudo
2. Fedeltà sonora
3. Probabilità di malfunzionamenti nel primo anno di utilizzazione
4. Dimensione del trasformatore per l'alimentazione
5. Ergonomia dei comandi
6. Linearità della risposta in frequenza

Quali di esse sono esterne e quali sono interne?

Esercizio 4: Assimilando le qualità di un programma alle proprietà delle automobili, il fatto che i motori di un certo modello possono essere montati su diversi modelli di automobile a quali qualità interne fa riferimento?

Risposta: Portabilità (il motore deve funzionare su macchine diverse), Comprensibilità (il meccanico, che non ha progettato la macchina, deve poter capire come funziona il motore per montarlo)

Esercizio 5: Considerare due qualità alla volta e valutare il loro rapporto reciproco. Sono in contrasto, una aiuta l'altra?

Risposta:

Part II

Introduzione a Java come Linguaggio OO:

1 Programma e Paradigmi di Programmazione:

Indipendentemente dal linguaggio di programmazione utilizzato:

$$\text{Programma} = \text{Oggetto} + \text{Operazioni}$$

Dove gli oggetti sono la rappresentazione delle informazioni (cioè dei dati) e le Operazioni sono manipolazioni di questa rappresentazione. Abbiamo tre paradigmi di programmazione:

- Imperativo: enfasi sulle operazioni come comandi che cambiano lo stato dell'elaborazione. Gli oggetti sono funzionali all'elaborazione.
- Funzionale: enfasi sulle operazioni come funzioni che calcolano risultati. Gli oggetti sono funzionali a questa elaborazione.
- Orientato agli Oggetti: qua l'enfasi è sugli oggetti, che rappresentano esplicitamente delle informazioni di interesse. Le operazioni qui sono strumenti che manipolano gli oggetti

Vediamo quindi che nei paradigmi Imperativo e Funzionale le operazioni sono il focus, e su queste applico gli oggetti, mentre nel paradigma OO (Object Oriented) il focus sono gli oggetti, e su questi applico le funzioni. Con ogni linguaggio si può lavorare con tutti e tre i paradigmi, ma in genere un linguaggio offre strumenti più sofisticati per un paradigma e meno per gli altri. Java ad esempio è un linguaggio ad alto livello Orientato agli oggetti (OO), ciò significa che è fatto per l'OOP, ma supporta bene anche i paradigmi imperativo e funzionale.

La Java Virtual Machine: La JVM rende Java portabile e indipendente dalla piattaforma, poiché un qualsiasi altro calcolatore, oltre a quello in cui è stato scritto, può eseguire un certo programma Java, se ha la JVM. Vedremo nella prossima sezione l'importanza della JVM

2 Particolarità di Java:

Un programma in Java è un insieme di classi Java (che sono composte da Oggetti e Operazioni, quindi questa definizione è in accordo con quella generale data all'inizio). Alcune osservazioni:

- La sintassi delle istruzioni è simile a quella di C.
- In Java una funzione non può contenere definizioni di altre funzioni
- La conversione automatica dei tipi è assente in Java.
- Boolean è un tipo predefinito.
- + è l'operatore predefinito di concatenazione tra le stringhe

3 Scrittura, Compilazione ed Esecuzione di Programmi in Java:

Vediamo come creare un programma Java, come compilarlo ed eseguirlo, Per adesso solo da riga di comando, poi vedremo anche da IDE. Abbiamo 3 fasi.

3.1 Preparazione del Testo del Programma:

Si usa un qualsiasi editor di testo per scrivere il *codice sorgente*, in accordo con la sintassi propria di Java. **Il file prodotto deve aver il nome della classe che è al suo interno, ed estensione .java.** Solitamente in un file Java c'è una sola classe, ma è possibile definire delle linear class, cioè classi interne, che mi permettono di definire più classi in un solo file; vedremo esempi di questo durante il corso.

3.2 Compilazione del Programma, generamento del ByteCode:

Un compilatore ora, a partire dal file .java, analizza (analisi sintattica) il codice sorgente e genera lo Java ByteCode, che è un file .class. Il compilatore standard è javac del JSRK (Java Standard Development Kit). Questi file .class sono il codice macchina per la JVM⁵ (Java Virtual Machine). Operativamente, per compilare si scrive, da riga di comando:

```
javac NomeFile.java
```

3.3 Esecuzione del programma, esecuzione del ByteCode:

Il file .class viene caricato nell'interprete JVM, che lo traduce nel codice macchina del tuo elaboratore e lo esegue. Questo passaggio attraverso l'interprete JVM permette di tradurre uno stesso programma e farlo funzionare per elaboratori diversi, rendendo quindi Java *portable*. Quando avviamo un programma la JVM va a cercare un metodo main, questo è il primo ad essere eseguito. Eventuali altri file class vengono caricati a run time. Cioè c'è un collegamento dinamico: Java carica in memoria la classe quando viene utilizzata, non prima. Il parametro di configurazione della JVM che le dice dove andare a cercare le classi è chiamato *ClassPath*. Già alla scrittura del codice Java identifica la maggior parte degli errori, quindi a tempo di compilazione non a run time. Questo perché Java è un linguaggio abbastanza "ingabbiato", il ché riduce errori a run time. Ciò non vuol dire che non possano esserci errori a tempo di esecuzione, ad esempio il compilatore non si accorgerà mai di un ciclo infinito.

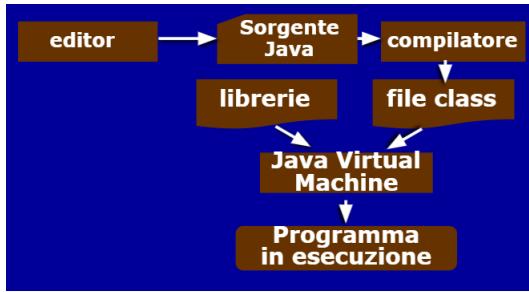


Figure 2: Riassunto delle fasi di preparazione del testo, compilazione e esecuzione

Bene, facciamo un esempio, analizziamolo e poi vediamo le tre fasi appena spiegate come gli si applicano.

3.4 Facciamo un esempio:

```
import java.lang.*;
public class Primo {

    public static void
        main(string[] args){
            system.out.println("JAVA");
        }
}
```

Spiegazione del codice:

- All'inizio si importa la libreria java.lang tramite il comando `import java.lang.*;`. Questo comando è in realtà inutile perché il package è già importato di default.
- Definiamo una *classe* con `public class`, e la chiamiamo Primo. Questa classe contiene il *metodo main* che, come in C, è fondamentale per far partire il programma: è il metodo di avvio del programma. A differenza di C però, c'è solo un parametro ed è obbligatorio: diamo in input

⁵Attenzione quindi! Si tratta del codice macchina della JVM, non dell'elaboratore su cui stai lavorando!

un array di stringhe chiamato args⁶. Questo parametro è anche chiamato *Java Command Line Arguments*.

- All'interno del metodo main troviamo in questo caso `System.out.println("JAVA");`. System è una classe che contiene variabili statiche, da cui noi prendiamo solo la variabile out, out ci dà accesso allo standard output di Java, cioè ci permette di stampare a schermo. Utilizziamo allora `println` per stampare "JAVA" a schermo. Abbiamo praticamente fatto un Hello World Program.
- Adesso vediamo l'utilità di quel parametro args: possiamo ciclare su tutto ciò che c'è dentro args e andare a stamparlo. Per fare ciò usiamo il ciclo for, identico a C:

```
for (int i = 0; i < args.length; i++) {
    System.out.println(args[i])
}
```

E i varia da 0 fino all'ultimo argomento di args. Proviamo a farlo da linea di comando, compiliamo ed eseguiamo il codice, ma alla fine del comando di esecuzione, aggiungiamo delle stringhe, separate da uno spazio, vedremo che, per il codice che abbiamo scritto qua sopra, Java stampa le stringhe in ordine una sotto l'altra.

Nota: questo è un programma che ben si addice a un paradigma di programmazione procedurale. Java lo può fare perché gli OOP permettono comunque la programmazione procedurale.

Le tre fasi di un programma in questo esempio:

Preparazione Testo: Scritto il testo, poiché all'interno del file è definita la classe *pubblica* Primo, dovremo chiamare il file Primo.java. Il fatto che devo chiamare il file come la classe pubblica ci fa capire che in un file ci può essere solo una classe public (*ma molteplici altre classi senza modificatore di visibilità*⁷), e che essa è obbligatoria.

Compilazione: Primo.java è il mio file. Da riga di comando, ci posizioniamo nella directory dove è contenuto il file⁸ e scriviamo

```
javac Primo.java
```

Per compilare. Questo mi produce il file Primo.class.

Esecuzione: Adesso ho il mio file Primo.class. Sempre da riga di comando scrivo:

```
java Primo
```

⁶In Java gli array sono delle classi che hanno già di loro attributo lenght, quindi non ho bisogno del parametro aggiuntivo che davo in C per la lunghezza dell'array.

⁷Le classi, vedremo dopo, possono avere solo il modificatore public o non averlo.

⁸Nella VM da esame per muoversi tra directory si utilizza il comando cd seguito dal comando relativo a "vai avanti" o "vai indietro". Quando si apre il terminale bisogna andare avanti verso il desktop, quindi cd -P, e poi li dipende se il file è in un'altra directory o no.

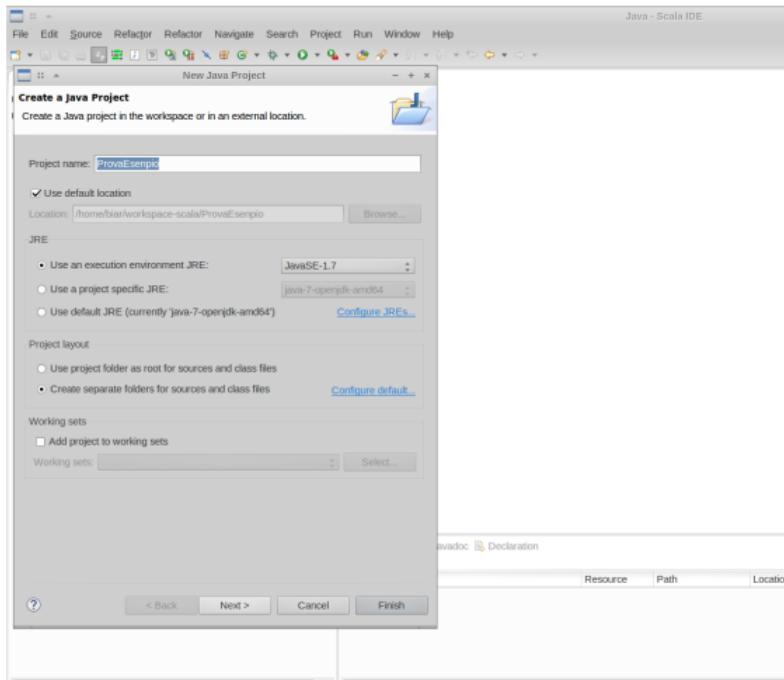
4 IDE:

Per il programma precedente abbiamo utilizzato un classico *editor* di testo (anche il semplice blocco note di Windows è un editor di testo), e poi compilato da riga di comando. Per programmi più complessi e grossi compilare da riga di comando diventa un inferno, allora si utilizzano gli Integrated Development Environment (IDE), che permettono una gestione migliore del codice e delle classi. Gli IDE, forniscono suggerimenti, debugging semplice e dispiegamenti⁹ automatici sui server. Un IDE è composto da un editor per codice sorgente, un compilatore e un interprete, e un debugger.

4.1 Esempio con l'IDE Eclipse:

Avviamo eclipse. In eclipse tutti gli elementi di un programma (sorgente, file.class, package ecc.) sono contenuti in un progetto, che possiamo pensare come un'applicazione. Quindi andiamo a creare un nuovo progetto facendo:

File -> New -> Java Project

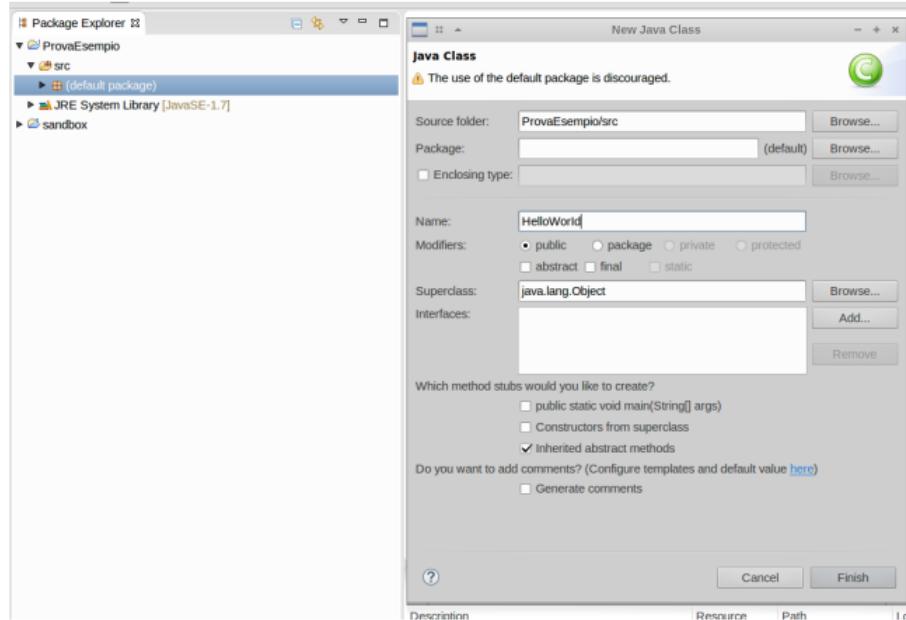


Bisogna specificare la versione di Java da utilizzare. Per evitare problemi di qualsiasi tipo però noi possiamo cliccare su *Use default JRE* e poi *next*. Facciamo *finish*. Quello che è successo è che a sinistra è stata creata una cartella, il nostro progetto. Vediamo che questo contiene una directory *src* (source, sorgente) e *JRE System Library*. Clicchiamo col tasto destro su *src*, poi *New¹⁰*, *Class*. Qui siamo pronti per creare la classe *HelloWorld*.

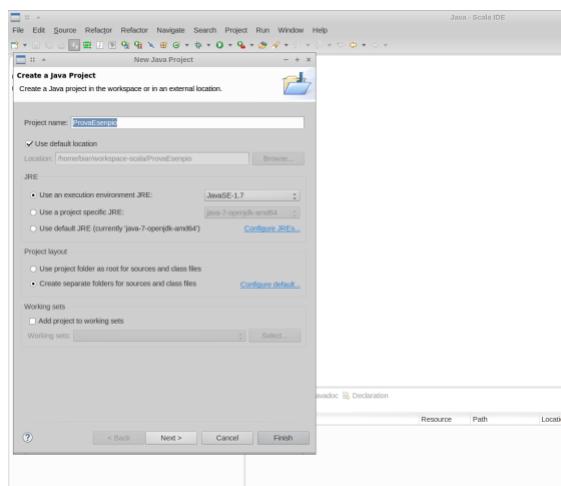
⁹deployment, o ancora un altro sinonimo, distribuzione, vediamo questo concetto più avanti

¹⁰Notare che avendo scelto il *default package*, che coincide con il package vuoto, la classe non sarà in un package. Come si vede in alto nella schermata di creazione di una classe ci dà il warning *L'uso del default package è scoraggiato*. Questo effettivamente può rientrare nella cosiddetta *cattiva programmazione*, sarebbe buona prassi che sotto *source folder*, scrivessimo un *package* in cui poi si troverà la nostra prima classe.

Adesso creiamo la classe: Sempre nella schermata di creazione della classe possiamo decidere



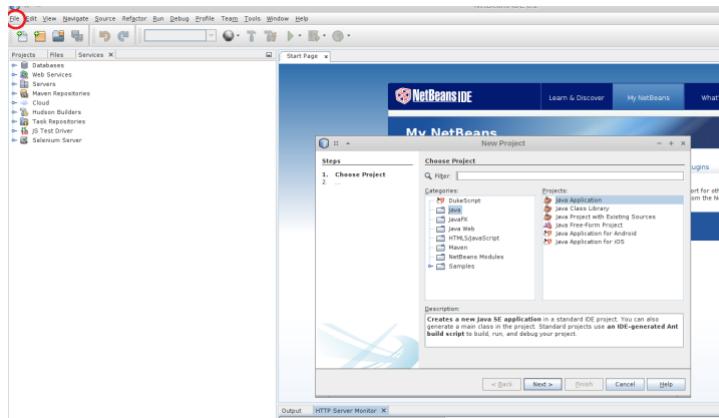
di far creare a Eclipse il metodo main, spuntando `public static void main(String[] args)`. Scelte le nostre opzioni, premiamo finish. Ora dobbiamo scrivere il codice sorgente nella nostra classe. Scritto il codice basta cliccare run in alto a destra e il programma viene eseguito in basso sulla console di Eclipse.



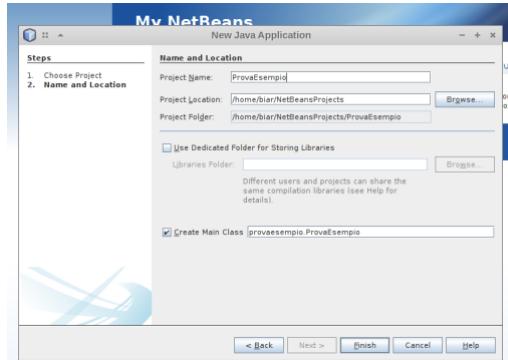
4.2 Esempio con l'IDE NetBeans:

Riproponiamo lo stesso esempio ma con NetBeans.

File -> New Project -> Java -> Java Application



Come in Eclipse posso spuntare *create main class* per far creare all'IDE una classe main messa nel package che stiamo creando. Sulla sinistra abbiamo la nostra applicazione, che, come in Eclipse, ha dentro di sé le *Libraries* e *Source Packages*. A questo punto vado a creare una nuova classe java che chiamo hello world. Sarebbe *buona prassi mettere i nomi dei package senza underscore e tutti in minuscolo*.



Vediamo che NetBeans crea dei commenti all'inizio della classe, in alcuni casi utili, ad esempio @author. Le indicazioni che iniziano per @ sono indicazioni per la produzione automatica di documentazione relativa al codice, e si chiamano annotazioni. Possiamo dare delle indicazioni che servono per creare la documentazione tramite uno strumento che si chiama java doc. Se cerchiamo java doc su google troviamo tutte le pagine di documentazione di Java. Torniamo all'IDE, in alto c'è il pulsante di play che esegue il codice.

Nota: Netbeans mette i metodi e le variabili static in corsivo.

Nota II: su NetBeans puoi selezionare un testo indentato male, premere il tasto destro e poi format per indentarlo.

Nota III: *debugger in Netbeans*. Parliamo dei **breakpoints**: punti in cui, una volta che il programma arriva lì, si blocca e il controllo viene passato all'IDE, o qualunque altro programma che gestisce il debugging. Come si attiva un breakpoint? Basta cliccare due volte sul numero a sinistra relativo alla riga di codice dove posizioniamo il breakpoint. Bene, facciamo ora **debug file** sulla classe e vediamo che il breakpoint diventa verde. Passando col mouse e cliccando sul pop-up possiamo vedere sulle variabili che valore hanno assunto codeste, così *possiamo vedere lo stato del programma in un certo momento*. Da qui posso far andare avanti il programma con **continue** in

alto; a destra di continue ci sono dei tasti che permettono di andare passo passo per ogni riga di codice del programma, così da vedere lo stato del programma in ogni momento: `step over` va alla riga sottostante, mentre `step into`, nel caso in cui si sia in una chiamata di metodo, porta direttamente alla prima riga di codice del metodo (potresti dover cliccare sul nome del metodo in cui vuoi entrare, dopo aver premuto `step into`).

5 API Java:

Con API (Application Programming Interface) si indicano tutta una serie di procedure che sono a disposizione del programmatore. In particolare le API Java, anche chiamate librerie Java, mostrano le classi e descrivono i metodi al loro interno. Le API per Java 8¹¹ sono disponibili al seguente URL:

<https://docs.oracle.com/javase/8/docs/api/>

Vediamo che per ogni classe le API riportano e descrivono superclassi, sottoclassi, funzionalità, campi, costruttori e metodi. La java doc sarà disponibile anche all'esame

Nota: Deprecated è un metodo o un costruttore che verranno eliminate prima o poi in versioni future della libreria.

Nota II: All implemented interface: non possiamo capire bene ancora cosa sia ma per ora facciamo un esempio. Vediamo che nella classe Scanner di `java.util`, *All implemented interface* è seguito da *closeable* e *autocloseable*. Closeable ci indica che è possibile chiudere uno stream, un metodo di closeable è `close()`, che applicato su un'istanza di Scanner chiude quello stream di dati. Autocloseable invece ci dice che non appena viene distrutta una classe (cioè quando non ci sono più riferimenti a lei, esempio fine del programma grazie al GC, lo stream viene chiuso). Perché chiudere lo stream di dati? Si tratta di una buona prassi andare a chiudere le risorse, perché magari ho uno stesso thread che cerca di leggere dallo stesso stream e questo può creare problemi, soprattutto in progetti grossi.

6 Dispiegare programmi in Java:

Un programma in java viene distribuito come un file Java Archive: `.jar`. Può essere creato da riga di comando con il comando:

```
jar cf <nomeArchivio> <file/package>
```

O dal vostro IDE. Per esempio in Eclipse si fa l'export del progetto con il tasto destro, avendo cura nelle varie finestre del *Wizard*¹² di indicare la classe con il main, così che venga creato correttamente il MANIFEST. In NetBeans invece si fa build&deploy, sempre accessibile con tasto destro

¹¹(Ottobre 2021) Java è alla versione 16, ma i cambiamenti non sono così raggardevoli da rendere obsolete le API per Java 8.

¹²Cioè il procedimento a passi successivi che state eseguendo grazie alle finestre che compaiono, chiamato così perché rende un procedimento tecnicamente difficile molto facile all'utente grazie alle finestre a passi successivi, come appunto, per magia

Part III

Le Classi in Java:

1 Che cos'è una classe:

Le classi servono a rappresentare classi di oggetti in Java. Sono una forma estesa dei record, presenti nei linguaggi tradizionali, perché non servono soltanto per memorizzare dei dati ma contengono anche *le operazioni* per manipolarli. In java i nomi di classe iniziano con la lettera maiuscola, mentre i nomi di variabili con la minuscola (questo è lo standard di programmazione). Una classe Java è caratterizzata:

- Nome: la identifica univocamente nel programma
- Campi dati: per memorizzare informazioni degli oggetti, cioè delle sue istanze. Le variabili contenute qui sono le variabili di istanza e le variabili static.
- Campi funzione: che definiscono le operazioni che è possibile invocare sugli oggetti sue istanze. Qui sono contenuti i metodi, anche quelli static.
- Modificatori di visibilità: definiscono quali campi sono visibili e quali nascosti ai client

Facciamo un esempio: Vogliamo scrivere una classe per rappresentare persone. Gli oggetti quindi sono le persone. Le caratteristiche di interesse per le persone sono il nome, definito una volta sola e immodificabile, e la residenza, sempre modificabile. Ecco l'implementazione:

```
public class Persona {  
    //campi dati (variabili di istanza)  
    private String nome;  
    private String residenza;  
  
    //campi funzione (metodi)  
    public String getNome() {  
        return nome;  
    }  
    public String getResidenza() {  
        return residenza;  
    }  
    public void setResidenza(String nuovaResidenza) {  
        residenza = nuovaResidenza;  
    }  
}
```

Osservazioni: Il nome della classe è Persona. I campi dati sono nome e residenza, entrambi di tipo String ed entrambi private, quindi non accessibili ai client. I campi funzione sono getNome(); getResidenza() e setResidenza() e sono tutti public, quindi accessibili ai client.

2 Come si usa una classe?

Un client può usare la classe Persona per generare oggetti di tipo Persona. Esempio:

```
public class ClienteClassePersona {  
    public static void main(String[] args) {  
        Persona p1; //dichiara un riferimento a un oggetto persona  
        p1 = new Persona(); //alloca memoria e la assegna al riferimento  
        p1.setResidenza("Roma"); //imposta la residenza a Roma  
        System.out.println(p1.getResidenza()); //stampa la residenza, quindi "Roma"  
    }  
}
```

2.1 Riferimenti e Allocazione:

In Java non si possono definire variabili di tipo oggetto ma solo riferimenti a oggetto, come nell'esempio sopra. Quindi i riferimenti sono assimilabili ai tipi base. In java non si può creare specifica dimensione di memoria (come in C con la malloc) ma solo lo spazio per gli oggetti. Abbiamo:

- Allocazione Statica: decisa a tempo di compilazione ed effettuata nello stack
- Allocazione Dinamica: decisa a tempo di esecuzione con l'unico comando disponibile per questo: new, ed effettuata nell'heap

Facciamo un esempio:

```
Persona p1;  
p1 = new Persona();  
p1.setResidenza("Roma");  
Persona p2;  
p2 = p1  
int i1 = 5  
int i2 = i1
```

Vediamo cosa succede in memoria:

- Persona p1;

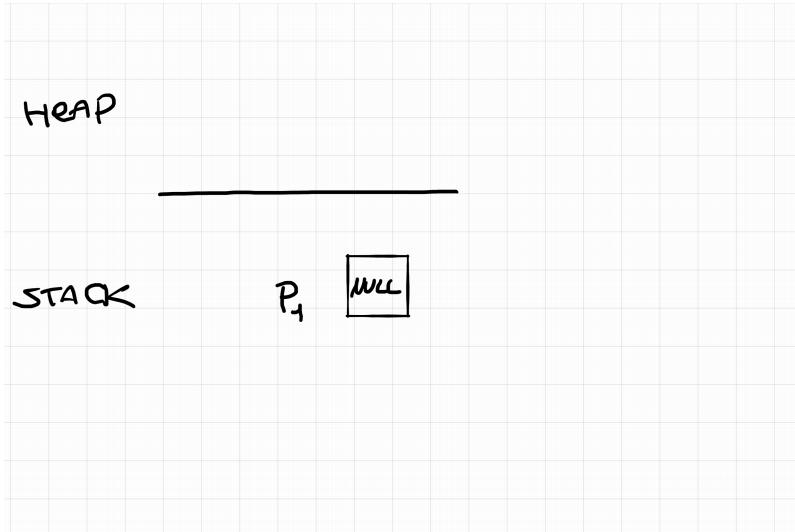


Figure 3: Viene allocata staticamente memoria per il riferimento all'oggetto persona.

- p1 = new Persona();
- p1.setResidenza("Roma");
- p2 = p1
- int i1 = 5; int i2 = i1

Deallocazione degli Oggetti, Garbage Collector: Come eliminare gli oggetti che non si usano più, per evitare di saturare la memoria? In C++ se ho qualcosa in memoria e non lo voglio devo usare free o delete, in Java invece c'è un vantaggio (ma ogni vantaggio rende il linguaggio più lento) su questo, perché la deallocazione dall'heap è effettuata in maniera automatica. Un oggetto viene eliminato quando non ci sono più riferimenti che fanno riferimento all'heap (non immediatamente perché il GC gira in maniera asincrona rispetto al flusso del programma). In C questo non ha senso perché con l'aritmetica dei puntatori posso raggiungere una zona che non era accessibile perché non aveva più puntatori. Se si prova a usare un metodo su un qualcosa su cui ha agito il GC si ottiene un errore, riprendendo l'esempio di prima, se aggiungo le ultime due linee di codice:

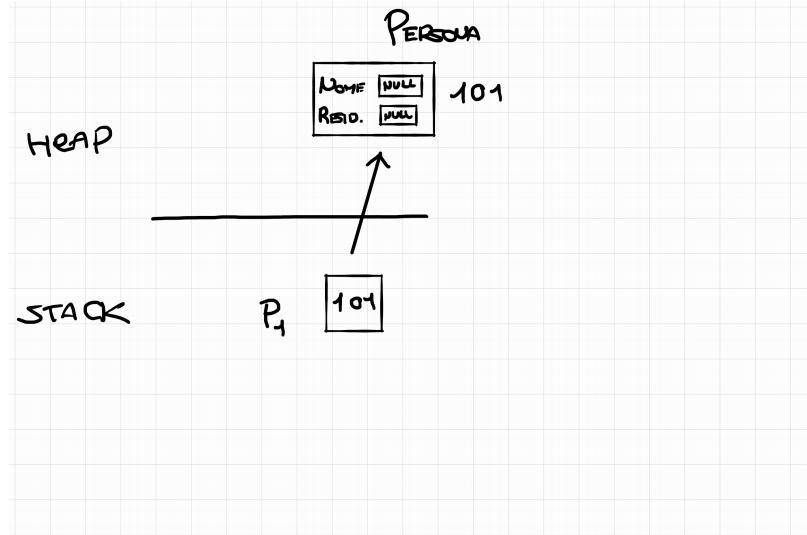


Figure 4: Viene allocata dinamicamente memoria per un istanza della classe persona. L'indirizzo di questa memoria (che abbiamo messo convenzionalmente a 101) viene assegnato al riferimento

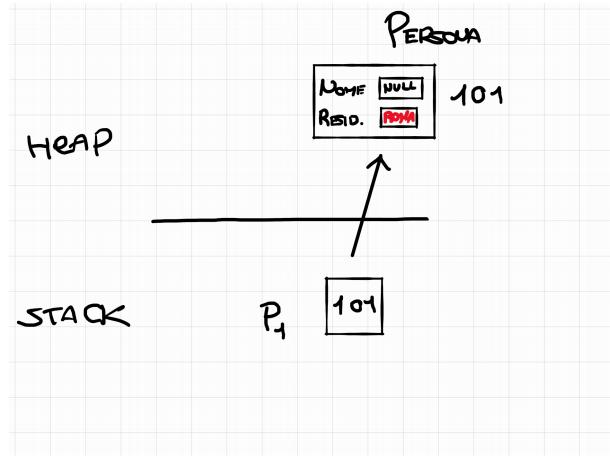


Figure 5: Utilizzando il metodo setResidenza sul riferimento ad oggetto p1, riesco a modificare la residenza dell'istanza

```

Persona p1;
p1 = new Persona();
p1.setResidenza("Roma");
Persona p2;
p2 = p1
int i1 = 5
int i2 = i1
p1 = null;
p2 = null;

```

Allora non potrò più accedere all'istanza di Persona, e provandoci, per esempio con `getResidenza()`, otterò l'errore:

```
null pointer exception
```

3 Campi Public e Private:

Public indica che il campo (sia dati sia funzione) è visibile ai client della classe. Private indica che il campo non è visibile ai client, perché è solo per la classe stessa. Tipicamente è buona pratica di programmazione:

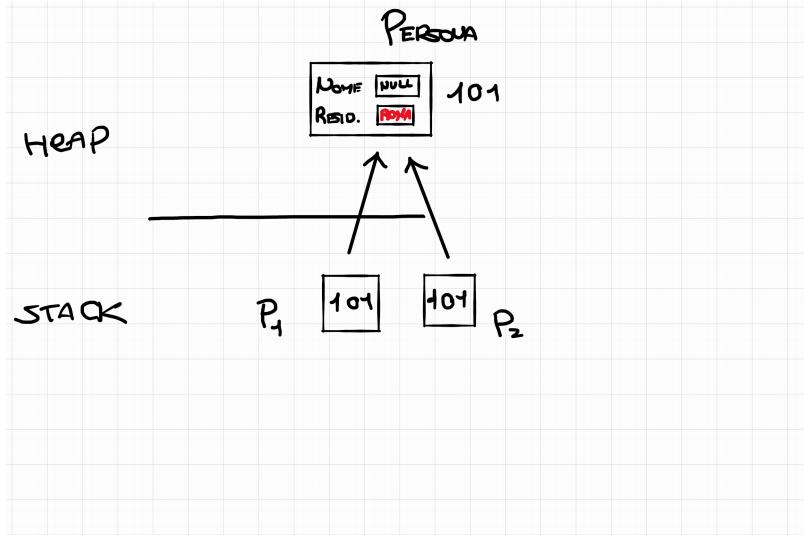


Figure 6: Dopo essere stata dichiarato, al riferimento p2 viene assegnato “=” lo stesso indirizzo di p1. Ciò significa che p2 e p1 si riferiscono alla stessa istanza (non a due copie uguali). Quindi modificando l’istanza con un riferimento, anche l’altro vedrà l’istanza modificata.

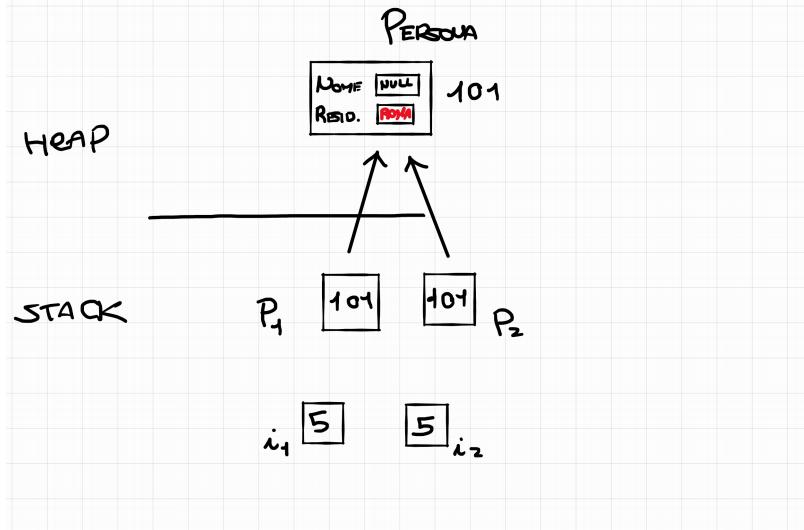


Figure 7: In questo caso si parla di assegnazione tra tipi predefiniti, non tra riferimenti, qua l’assegnazione “=” copia il valore contenuto in una variabile e lo inserisce nell’altra. Essendo questa una copia, se modifichiamo una variabile l’altra non vedrà la modifica, ma rimarrà come era prima.

- Dichiarare **public** i metodi della classe che servono ai clinet. Questi sono le funzionalità che il modulo propone all'esterno, cioè i lollipop dell'interfaccia. Per questo motivo l'insieme dei campi **public** è l'interfaccia pubblica della classe. Ricordati l'immagine del modulo.
- Dichiarare **private** i campi dati e le funzioni ausiliarie che servono solo nel modulo per la realizzazione delle funzionalità.

Non sono solo **public** e **private** i livelli di accesso ma per ora ci bastano,

Facciamo un esempio: se il nostro server è la classe **Persona**:

```
public class Persona {  
  
    //campi dati (variabili di istanza)  
    private String nome;  
    private String residenza;
```

```

//campi funzione (metodi)
public String getNome() {
    return nome;
}
public String getResidenza() {
    return residenza;
}
public void setResidenza(String nuovaResidenza) {
    residenza = nuovaResidenza;
}
}

```

E il nostro client è:

```

public class ClienteClassePersona2 {
public static void main(String[] args) {
    Persona p1;
    p1 = new Persona();
    p1.setResidenza("Roma");
    //OK il campo setResidenza e' public
    System.out.println(p1.getResidenza());
    //OK il campo getResidenza e' public
    System.out.println(p1.residenza);
    //ERRORE il campo residenza e' private
}
}

```

Vediamo che il terzo tentativo di accesso provoca un errore a tempo di compilazione: *tutti gli errori di visibilità sono errori a tempo di compilazione.*

4 Overloading:

Possibilità di avere funzioni che hanno lo stesso nome ma con numero e tipo dei parametri in ingresso diversi (il tipo di ritorno non fa parte della segnatura, quindi non interessa, può cambiare o meno nelle funzioni overloaded sempre overloaded rimangono). Perché farlo? Ad esempio:

- Voglio fornire al client un metodo in una versione in cui ha più controllo sui parametri di ingresso e in una versione in cui ne ha meno. Quindi varia il numero di parametri in ingresso.
- Voglio fornire al client un metodo diverse versioni in cui, in ognuna di queste, dà in ingresso un tipo diverso. Quindi varia il tipo dei parametri in ingresso

In ogni caso però non posso avere due metodi che hanno stesso nome e parametri di ingresso ma variano solo per il tipo di ritorno: ci deve essere differenza nei parametri di ingresso. L'overloading è utile per i costruttori che vediamo dopo.

Facciamo un esempio:

```

public class Overloading {

    private int x;

    public void f() { //OK
        x++;
    }

    public void f(int n) { //OK
        x = n;
    }

    public void f(int m, int n) { //OK
        x = m*n;
    }

    public int f() { //ERRORE di compilazione
        return x;
    }
}

```

```
}
```

5 Costruttori:

Riprendiamo la classe persona:

Facciamo un esempio: se il nostro server è la classe Persona:

```
public class Persona {  
  
    //campi dati (variabili di istanza)  
    private String nome;  
    private String residenza;  
  
    //campi funzione (metodi)  
    public String getNome() {  
        return nome;  
    }  
    public String getResidenza() {  
        return residenza;  
    }  
    public void setResidenza(String nuovaResidenza) {  
        residenza = nuovaResidenza;  
    }  
}
```

Per cambiare la residenza abbiamo setResidenza(), mentre, poiché non vogliamo che un client cambi nome ad una persona quando vuole, non abbiamo definito setNome(), ma allora come assegnamo la prima volta un valore al campo dati nome? Definendo un opportuno costruttore. Nel nostro caso:

```
public class Persona {  
    ...  
    public Persona(String n, String r) {  
        nome = n;  
        residenza = r;  
    }  
    ...  
}
```

anche per loro è ammesso l'overloading, ad esempio possiamo fare:

```
public class Persona {  
    ...  
    public Persona(String n, String r) {  
        nome = n;  
        residenza = r;  
    }  
    ...  
  
    public Persona(String n) {  
        nome = n;  
    }  
}
```

Quindi avendo due diversi costruttori, uno che mette sia nome che residenza, e uno che mette solo il nome. Bene adesso utilizziamoli: i costruttori vengono invocati con la parola chiave new¹³, quindi nel nostro caso potremmo fare:

¹³Quando non si sono definiti costruttori esplicitamente new utilizza il costruttore standard, invocato senza parametri, che lascia tutti i campi dati al loro valore di default. Qualora sia definito un qualsiasi altro costruttore, questo viene inibito, quindi non può essere più utilizzato.

```

    Persona p1 = new Persona("Luigi Bianchi"); //invociamo costruttore nome
    Persona p2 = new Persona("Giovanni Verdi", "Roma"); //invociamo costruttore
        nome-residenza
    System.out.println(p1.getNome()); //stampa "Luigi Bianchi"
    System.out.println(p2.getNome()); //stampa "Giovanni Verdi"

```

Abbiamo finalmente il codice completo della classe Persona:

```

public class Persona {
//campi dati (variabili di istanza)
private String nome;
private String residenza;
//costruttori
public Persona() {
    nome = "Mario Rossi";
    residenza = null;
}
public Persona(String n) {
    nome = n;
    residenza = null;
}
public Persona(String n, String r) {
    nome = n;
    residenza = r;
}
//campi funzione (metodi)
public String getNome() {
    return nome;
}
public String getResidenza() {
    return residenza;
}
public void setResidenza(String nuovaResidenza) {
    residenza = nuovaResidenza;
}
}

```

6 Inizializzazioni Implicite dei campi dati di una classe:

Nel caso in cui non si assegna un valore ad un campo dati nel momento in cui viene dichiarato, queste sono i valori che prendono di default¹⁴:

Tipo	Inizializzato a:
int	0
float, double	0.0
char	'\0'
boolean	false
class C	null

Nota: Queste inizializzazioni si applicano alle variabili di istanza delle classi, ma NON sono effettuate per le variabili locali delle funzioni, che se non inizializzate scatenano un errore al primo utilizzo. Facciamo un esempio:

```

public class MainApp {
    public static void main(String[] args) {
        int a;
        System.out.println(a);
    }
}

```

¹⁴si noti la differenza con C, dove se l'inizializzazione non era esplicita il valore del campo era indeterminato, poiché era il valore sporco assunto da qualche altra variabile che prima aveva riempito quella zona di memoria.

}

Questo programma definisce una variabile senza inizializzarla. Compilando compare un *errore a tempo di compilazione*: variable "a" might not have been initialized.

7 Esercizio:

Realizzare una classe Java per rappresentare automobili. Delle automobili interressano targa, modello colore e persona proprietaria. Targa e modello non sono modificabili ma il resto sì. Utilizziamo questo algoritmo per creare la classe:

- Prima decidiamo come rappresentare gli Automobile. Quindi definiamo i campi dati (private per buona prassi e stavolta necessità, dato che targa e nome non devono essere modificati).
- Poi scegliamo l'interfaccia della classe. Quindi realizziamo la segnatura dei metodi (public)
- Poi realizziamo i metodi, che prima avevamo solo segnato.

Ecco qua il risultato:

```
public class Automobile {  
  
    //campi dati privati per la  
    //rappresentazione degli oggetti automobile  
  
    private String targa;  
    private String modello;  
    private String colore;  
    private Persona proprietario;  
  
    //campi funzione pubblici per:  
    //costruttori  
  
    public Automobile(String t, String m, String c) {  
        targa = t; modello = m; colore = c; proprietario = null;  
    }  
    public Automobile(String t, String m, String c, Persona p) {  
        targa = t; modello = m; colore = c; proprietario = p;  
    }  
  
    //funzionalita' pubbliche della classe  
  
    public String getTarga() {  
        return targa;  
    }  
    public String getModello() {  
        return modello;  
    }  
    public String getColore() {  
        return colore;  
    }  
    public void setColore(String nuovoc) {  
        colore = nuovoc;  
    }  
    public Persona getProprietario() {  
        return proprietario;  
    }  
    public void setProprietario(Persona nuovop) {  
        proprietario = nuovop;  
    }  
}
```

Importante! Persona e Automobile devono essere nella stessa cartella!

7.1 Esercizio, Client con due Metodi Static:

realizzare un Client del Server Automobili, che ha queste due funzionalità (metodi):

- Client verniciatura, prende come parametri un'automobile a e un colore c e cambia il colore di a in c
- Client immatricolaFerrari360GT, prende come parametri una targa e restituisce una nuova automobile del modello e colore prestabilito.

Vediamo che entrambi i metodi sono static: non sono relative a una particolare istanza della classe, quindi non vengono invocate su un oggetto della classe ma sulla classe direttamente. Ecco la realizzazione del Client:

```
public class ServiziAuto {  
  
    public static void verniciatura(Automobile auto, String colore) {  
        auto.setColore(colore);  
    }  
  
    public static Automobile immatricolaFerrari360GT(String tar) {  
        return new Automobile(tar, "Ferrari360GT", "Rosso Maranello");  
    }  
}
```

Adesso creiamo una classe Main, che utilizza la classe Automobile e anche la classe ServiziAuto:

```
public class Main {  
  
    public static void main(String[] args) {  
        Automobile a = new Automobile("313", "Cinquecento", "Rosso e Blu");  
        stampaDatiAuto(a);  
        Persona p = new Persona("Paperino", "Paperopoli");  
        a.setProprietario(p);  
        stampaDatiProprietario(a);  
        ServiziAuto.verniciatura(a, "Rosso Maranello");  
        stampaDatiAuto(a);  
        Automobile b = ServiziAuto.immatricolaFerrari360GT("131");  
        stampaDatiAuto(b);  
        Persona c = new Persona("Clarabella", "Topolinia");  
        b.setProprietario(c);  
        stampaDatiProprietario(b);  
    }  
    //metodi ausiliari  
    private static void stampaDatiAuto(Automobile a) {  
        System.out.println("Automobile: " + a.getTarga() + ", "  
            + a.getModello() + ", "  
            + a.getColore());  
    }  
    private static void stampaDatiProprietario(Automobile a) {  
        System.out.println("Proprietario: "  
            + a.getProprietario().getNome() + ", "  
            + a.getProprietario().getResidenza());  
    }  
}
```

Notare come abbiamo usato il metodo stampaDatiAuto prima di definirlo. In Java questa cosa si può fare.

8 Il riferimento This:

Noi sappiamo che in un qualunque campo funzione non static di una classe possiamo far riferimento alle variabili di istanza o ai metodi di istanza della classe senza scrivere nulla prima del punto. Ci sono dei casi in cui ho necessità di dire esplicitamente che mi sto riferire alle variabili o ai metodi di istanza. Per questi casi c'è il this. This restituisce un riferimento all'oggetto che viene invocato su lei. Quali sono questi casi in cui ho necessità?

- Quando, specialmente nei costruttori, si ha una variabile locale o un parametro in ingresso della funzione con lo stesso nome di una variabile di istanza della classe¹⁵, è necessario usare il this. Esempio, se ho targa e modello come variabili di istanza:

```
public Automobile(String targa, String modello) {

    this.targa = targa
    this.modello = modello

    //se sto in un blocco di codice l'ultima definizione che ho fatto di un nome
    //di variabile e' quella valida. Quindi targa definita come parametro mi
    //nasconde targa definita come variabile di istanza. Necessito quindi di
    //usare this.
}
```

- Sempre, per leggibilità: conviene sempre usare il this per riferirsi alle variabili di istanza della classe, così da rendere più leggibile il codice e far capire subito cosa fa.

9 Esercizi:

Realizzare una classe Java per rappresentare punti nello spazio cartesiano:

```
//File Punto.java

public class Punto {

    //campi dato
    private double x;
    private double y;
    private double z;

    //costruttori
    public Punto(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public Punto() {
        x = 0;
        y = 0;
        z = 0;
    }

    //funzionalità pubbliche della classe
    public double getX() { return x; }
    public double getY() { return y; }
    public double getZ() { return z; }
}
```

Realizzare una classe Java per rappresentare segmenti nello spazio cartesiano:

```
//File Segmento.java

public class Segmento {

    //campi dato
    private Punto inizio;
```

¹⁵Sembra cattiva programmazione, ma specialmente nei costruttori si fa per leggibilità.

```

private Punto fine;

//costruttori
public Segmento(Punto inizio, Punto fine) {
    this.inizio = inizio;
    this.fine = fine;
}

public Segmento(Punto fine) {
    this.inizio = new Punto();
    this.fine = fine;
}

//funzionalita' pubbliche della classe
public Punto getInizio() { return inizio; }
public Punto getFine() { return fine; }

}

```

10 Package:

Le classi in Java sono ordinate in cartelle e sottocartelle, che corrispondono¹⁶ ai package. Non basta solo la divisione in directory ma bisogna esplicitamente scrivere il package nel file Java per dire che le classi contenute appartengono a quel package, attraverso il comando (è proprio il primo del file) `package mio_package`. Ogni classe è in uno e un solo package; se esso non è specificato, la classe è nel package vuoto, quello di default. I package hanno tre funzioni:

- Pulizia e ordine
- Isolare il codice da altro codice indesiderato
- Limitare la visibilità dei campi dati e funzione. In particolare con il livello di accesso non qualificato (v. sotto)

Un esempio di package sono le librerie Java, alcune già importate di default (come `javalang`, che contiene la classe `System`). Per utilizzare una classe definita in un package ci sono due modi:

- Uso "il nome lungo" (me lo so mezzo inventato sto nome) di fronte ad ogni dichiarazione e creazione di istanza. Quindi, ad esempio, se voglio istanziare `c`, istanza della classe `Classe_Da_Usare`, contenuta nel package `mio_package`:

```
nome_package.Classe_Da_Usare c = new mio_package.C();
```

Poi, quando vado a usare questa variabile, non ho più bisogno di scrivere questo davanti e posso usarla normalmente. Il package serve in fase di definizione della variabile e in fase della creazione dell'istanza.

- Importo, una sola volta e per tutte le classi del file, il package. Scrivendo, subito dopo l'eventuale comando `package mio_package` per inserire il file in un package, il comando:

```
import nome_package.Classe_Da_Usare;
```

Che è quindi nella seconda riga del file. Altrimenti, si possono importare tutte le classi di un package con il seguente comando¹⁷:

```
import nome_package.*;
```

¹⁶La corrispondenza tra cartelle e package si esprime in questo: tutti i file relativi al package `package`, devono risiedere in una directory (che coincide appunto col package) chiamata `package`. Questo significa che se il package è `package.sottopackage.sottosottopackage`, il percorso di cartelle per arrivare a un file che contiene una classe di questo package sarà - utilizzando il linguaggio dei filepath - `package/sottopackage/sottosottopackage/file.java`

¹⁷Per leggibilità, sempre meglio avere import esplicativi e non usare `*`. In ogni caso Java non importa tutte le classi del package, ma le mette in memoria solo se le utilizzi. Questa associazione dinamica di cui abbiamo già parlato viene chiamata *late binding*.

Questi metodi vanno usati a meno che non si stia cercando di accedere ad una classe di un package da una classe appartenente allo stesso package, in quel caso si può usare la classe direttamente.

10.1 Sottopackage:

Servono soltanto per un'organizzazione concettuale, le classi nel sottopackage non hanno nulla di speciale rispetto a quelle nel package padre. Infatti se faccio `import.mio_pack.*` non importo il sottopack, ma solo le classi nel package padre, ed è ovviamente valido anche il viceversa. Dal punto di vista funzionale quindi non c'è nessun legame tra package e sottopackage, sono package **indipendenti**. Nota che anche se importiamo un package, non possiamo riferirci ad un suo sottopackage semplicemente invocando il sottopackage, ma dobbiamo invocare anche il package importato, esempio¹⁸:

```
/File mio_package/C.java

package mio_package;

public class C {
    public void F_C() {
        System.out.println("Sono F_C()");
    }

    mio_package.mio_subpackage.D d =
        new mio_package.mio_subpackage.D();
}
```

Vediamo che per riferirci ad un'istanza d della classe D, contenuta nel sottopackage `mio_subpackage` del package `mio_package`, dobbiamo scrivere tutta la catena a partire da `mio_package`, anche se quest'ultimo è stato importato. Questo proprio perché package e sottopackage sono slegati. Non solo i sorgenti .java ma anche i class devono rispettare la stessa divisione gerarchica, ed essere divisi nelle cartelle e nelle sottocartelle che hanno dichiarato con il comando `package mio_package;`.

11 Livelli di accesso:

Cioè i livelli di visibilità di una classe, una variabile o un metodo.

- Livelli di accesso di una classe. Una classe può essere:
 - `public`: visibile anche al di fuori del package di appartenenza
 - `non qualificata`: quindi senza modificatore di visibilità. In questo caso è visibile solo alle classi del package a cui appartiene. Le classi con livello di accesso non qualificato sono classi d'ausilio.

Ricordiamo che un file .java può contenere una sola classe `public` e molteplici classi non qualificate, che solitamente sono d'ausilio a quella `public`. Il package che specifico nel file vale sia per la classe `public` che per le classi non qualificate.

Livelli di accesso per campi dati e funzioni:

- `public`: visibile da tutti i client
- `protected`: visibile dalla classe e dalle sue sottoclassi (cioè le classi derivate, v. dopo), anche se quest'ultime sono situate in altri package.
- `non qualificato`: visibile solo nelle classi del package a cui appartiene la classe in cui è stato definito il campo
- `private`: visibile solo nella classe in cui è stato definito il campo

Avendo messo i modificatori in ordine decrescente di visibilità.

Chiariamo i concetti con una tabella; Se io voglio chiamare il campo A dal campo B, quand'è che il campo B vede il campo A? Notare che non abbiamo inserito il caso di classe derivata nello stesso

¹⁸Attenzione che questo esempio è preso da una serie di esempi che partono dalla slide 12, e proprio in questa slide hanno sbagliato a scrivere i nomi dei file, perché hanno usato i punti `."` al posto degli slash `/` per indicare il filepath.

Il campo B è nella:	Public	Protected	No qual.	Private
stessa classe di A	SI	SI	SI	SI
classe dello stesso package di A	SI	SI	SI	NO
classe derivata di A in package diverso	SI	SI	NO	NO
classe non derivata di A in package diverso	SI	NO	NO	NO

package perché in questo caso si applica il regime della classe nello stesso package, cioè la seconda riga. Notare infine che anche i costruttori possono essere soggetti ai modificatori di accesso.

Information Hiding: Ma a che servono sti livelli di accesso? Per l'information hiding, principio cardine dell'OO. Questo principio dice che bisogna esporre all'esterno solo le informazioni che sono necessarie al client.

Accesso privilegiato e alto accoppiamento: Le classi nello stesso package hanno, per quanto appena detto, un livello di accesso privilegiato: quello rappresentato dal livello di visibilità non qualificato. In più, classi nello stesso package hanno alto accoppiamento; anche se, per i principi dell'OO noi vogliamo basso accoppiamento, le classi devono comunque lavorare insieme, quindi non posso avere accoppiamento pari a 0.

Osservazione finale sui package: Non esiste un metodo per impedire a una classe di dichiararsi appartenente a un package (voi potete quindi prendere una libreria Apache e dichiarare una vostra classe come membro di questa libreria). Questo è quindi un modo per aggirare i livelli di accesso. Ad esempio: creo una libreria che ha lo stesso package di quella di Apache e posso usare le variabili e i metodi non qualificati della libreria di Apache, che in realtà dovevano essermi nascosti. Ma perché questo? Perché la divisione tramite livelli di accesso non è fatta per sicurezza informatica o segretezza, ma solo per tenere il codice pulito e ordinato.

Part IV

Ereditarietà:

12 Derivazione tra Classi:

Se una classe astrae le caratteristiche generale degli oggetti, una classe derivata astrae le proprietà di un sottoinsieme degli oggetti¹⁹. Esempio: ho insieme Persona, la classe, che astrae le caratteristiche generali delle persone. All'interno di questo insieme ho il sottoinsieme Studente, classe derivata di Persona che astrae le caratteristiche peculiari degli studenti (quelle caratteristiche che distinguono gli studenti dalle persone) e, in più, eredita le caratteristiche delle persone. Infatti, *quando una classe derivata estende una classe eredita tutti i campi di quella classe, tranne quelli privati*. Esempio, così vediamo anche la sintassi:

```
public class B { //CLASSE BASE
    int x;
    void G() { x = x * 20; }
}

public class D extends B { //CLASSE DERIVATA
    void H() { x = x * 10; }
}
```

In questo esempio la classe derivata D ha un nuovo metodo, sua peculiarità, e in più eredita il metodo G e la variabile x (a meno che non siano dichiarate come *private*) dalla classe B, suo padre. *Si può derivare una classe derivata*. Si dice che la classe derivata specializza la classe base.

12.1 Principi Fondamentali della Derivazione:

- Principio 1: Tutte le proprietà definite per la classe base vengono implicitamente definite anche nella classe derivata, cioè vengono *ereditate*. Questo perché non voglio riscrivere tutte le caratteristiche della classe padre quando scrivo la classe derivata, ma solo le peculiari, quelle aggiuntive che distinguono la derivata dalla base.
- Principio 2: la classe derivata può avere ulteriori proprietà (quello di cui parliamo sopra).
- Principio 3: *ogni oggetto della classe derivata è anche un oggetto della classe base*. Questo significa che posso usare un oggetto della classe derivata in ogni situazione in cui posso usare un oggetto della classe base. In questi casi si dice che *la classe derivata è compatibile con la classe base*. Quindi se un metodo richiede come parametro un'istanza di una classe base io posso passargli la derivata. Questo vale anche per le variabili: posso inizializzare una variabile di un tipo base passando un'istanza di un tipo derivato. Esempio:

```
public class B {...} //classe base
public class D extends B {...} //classe derivata
...
B b = new D(); //POSSO FARLO! La classe D derivata e' compatibile con la classe
               B base.
```

- Principio 4: *Il viceversa non è vero*, cioè un oggetto della classe base NON è un oggetto della classe derivata (a meno che non lo castiamo, vedremo dopo), e non si può utilizzare un oggetto della classe base dove si può usare solo un oggetto della classe derivata. *Si dice che la classe base NON è compatibile con la classe derivata*, al contrario invece abbiamo visto prima che c'è compatibilità. Facciamo un esempio:

```
public class B {...} //classe base
public class D extends B {...} //classe derivata
...
```

¹⁹Anche se non lo scriviamo esplicitamente, tutte le classi di Java estendono una classe base: java.lang.Object

```
D d = new B(); //NON POSSO FARLO! La classe base B NON e' compatibile con la  
classe D derivata.
```

Vedremo come il *casting* risolverà questo problema.

12.2 Gerarchie di classi:

In profondità: Io posso definire delle gerarchie: partendo dalla base estendo e poi estendo la estesa. Le proprietà della seconda estensione sono le stesse della prima e della base + la roba specializzata. Questa è una gerarchia in profondità.

In ampiezza: Ma ci sono anche le gerarchie di classi in ampiezza: posso avere una base e poi 10 estensioni di questa che sono allo stesso livello di profondità. Quindi una classe può avere un numero qualsiasi di classi derivate.

Eredità Multipla: NON c'è ereditarietà multipla in Java: una classe ha solo una classe base. Una classe può estendere solo una classe base e non più di una. Specifica scelta progettuale di Java, non è che non si possa fare, infatti in C++ esiste l'ereditarietà multipla.

12.3 Casting:

Già lo conosciamo per i tipi di dato base. Ma nel caso di riferimenti a oggetto? Facciamo un esempio:

```
public class B {...}  
public class D extends B { int x_d; }  
...  
  
D d = new D();  
d.x_d = 10;  
B b = d; //OK b e d denotano lo stesso ogg.  
b.x_d = 20; //NO x_d non e' un campo di B!
```

Vediamo che abbiamo due classi: B e D che estende B aggiungendo un campo non qualificato x_d . Creiamo poi un'istanza D e la assegniamo a d. Su questa istanza invochiamo x_d . Dopo definiamo una variabile b e la inizializziamo con d, possiamo farlo perché la classe derivata è sempre compatibile con la classe base. Ora b e d referenziano lo stesso oggetto in memoria, soltanto che uno lo fa utilizzando una variabile definita di tipo b e l'altra di tipo d. Ora però, se provo a fare: $b.x_d = 20$ mi dà un *errore a tempo di compilazione*. Questo perché il **type checking** è **statico**²⁰, cioè viene fatto a tempo di compilazione. Il compilatore quando fa il parsing²¹ del codice sorgente si ritrova una variabile di tipo b, non si mette a ragionare e a vedere che l'oggetto referenziato a b è un'istanza di D. Questo però non è impossibile, possiamo avere riferimenti di diverse classi allo stesso oggetto, sotto alcune condizioni, ma allora come modificare il codice così da poter modificare x_d con il riferimento b? Lo facciamo con il casting.

```
(D)b.x_d = 20;
```

Adesso (D)b è un riferimento di tipo D e questo si può fare. In generale, se b non potesse diventare di tipo D, e lo usassimo in questo modo per forzare l'invocazione, il compilatore non direbbe nulla, si fiderebbe che noi sappiamo che b è di tipo D. Se poi effettivamente b non fosse di tipo D allora ci sarebbe un *errore a tempo di esecuzione*, non a tempo di compilazione però, il compilatore non controlla la veridicità del casting e si fida di noi. Ovviamente il compilatore si fida del programmatore se il casting è fatto tra classi lungo lo stesso cammino di una gerarchia (cioè devono essere in qualche modo collegate gerarchicamente), altrimenti riconosce che è impossibile che quel casting abbia senso. Quindi possiamo dire che:

- Il casting tra classi lungo lo stesso cammino in una gerarchia di derivazione è sempre *sintatticamente corretto*: nessun errore di compilazione e il compilatore si fida del programmatore.

²⁰Il type check è il controllare che i valori assegnati a una variabile siano di un tipo di dato ammissibile per il tipo della variabile

²¹il parsing è l'analisi sintattica del codice, volta a determinare la correttezza della grammatica formale

- Ma è responsabilità del programmatore che il casting sia anche *semanticamente* corretto, altrimenti ci saranno errori a tempo di esecuzione.

Vediamo un esempio di errore a tempo di esecuzione:

```

public class B {...}
public class D extends B { int x_d; }
...
B b = new B();
D d = (D)b; //OK a tempo di compilazione
d.x_d = 10; //Errore a tempo di esecuzione (ClassCastException) perche' x_d non esiste
            nell'oggetto a cui d fa riferimento, cioe' un'istanza di b, poiche' questo ha solo i
            metodi e le variabili del tipo B

```

Nota: Data una variabile possiamo verificare se questa è di un tipo specificato (lo vedremo più in là)

12.4 Livello di Accesso Protected:

permette ad una classe derivata, presente in qualsiasi package, di accedere a dei campi di una classe base. I campi protected sono molto usati, molto di più di quelli non qualificati che il professore definisce come zozzeria. Qual è il motivo dell'esistenza di questo livello di accesso? Il discorso è che una classe derivata, anche se è in un altro package, ha una relazione speciale con la classe padre: non è un client qualsiasi, poiché vogliamo poter utilizzare le istanze della derivata al posto di quelle della classe base, quindi necessitiamo di questo livello di accesso. Protected quindi apre metodi e variabili alle classi derivate (oltre che a tutte le classi dello stesso package del padre, come abbiamo già visto), ma blocca l'accesso ai clienti generici fuori dal package.

12.5 Costruttori delle Classi Derivate:

ovviamente anche le classi derivate possono avere i loro costruttori. Questi devono andare a inizializzare anche i campi della classe base (ricordiamo che le istanze della classe derivata hanno, oltre ai loro campi peculiari, anche tutti gli altri campi della classe base). Alcuni di questi campi della classe base però potrebbero essere private, come fa il costruttore di un'altra classe a inizializzarli? Java richiede in tal caso di invocare un costruttore della classe base nei costruttori della classe derivata, tramite il costruttore super(). Super è permette di accedere ai campi dati e funzione della classe immediatamente base (cioè il padre subito sopra la classe in cui viene invocata). Notare che *super()* deve essere la prima istruzione del costruttore. Vediamo degli esempi su tutte le casistiche possibili:

- Il costruttore della classe derivata contiene esplicitamente l'invocazione di un costruttore della base, tramite super()

```

public class B {

    private int x_b;

    public B(int a) { //costruttore della classe base
        x_b = a;
    }

    public int getXb() { return x_b; }
}

public class D extends B {

    private int x_d;

    public D(int b, int c) { //costruttore della classe derivata
        super(b); //invocazione esplicita del costruttore della classe base
    }
}

```

```

        x_d = c;
    }

    public int getXd() { return x_d; }
}

public class EsempioCostruttori1 {

    public static void main(String[] args) {
        D d = new D(3,4);
        System.out.println(d.getXb() + " " + d.getXd());
    }
}

```

Quindi la classe derivata, avendo più informazioni della classe base, con super(b) inizializza i campi della classe base e poi da sola inizializza i suoi peculiari. L'invocazione super(b) chiama il costruttore di B passandoli il parametro b, che nel nostro caso permette di inizializzare $x_b = b$. Questo non si poteva fare direttamente poiché x_b è private.

- Il costruttore della classe derivata NON contiene esplicitamente l'invocazione di un costruttore della base. In questo caso viene invocato automaticamente il costruttore di default (senza parametri) della base, e se questo è stato inibito o è stato ridefinito private si genera un errore a tempo di compilazione. In questo caso possiamo pensare di ridefinire un costruttore senza parametri esplicitamente, in modo che diventi lui il costruttore di default invocato quando non usiamo super. Facciamo un esempio:

```

public class B {

    ... //campi dati, funzioni e eventuali costruttori con parametri

    public B() { //costruttori senza parametri
        x_b = 10
    }
}

public class D extends B {

    ... //campi dati e funzioni

    public D(int c) { //costruttore della derivata, poiche' non c'e' super viene
                      //invocato il costruttore senza parametri della classe base, portando
                      //quindi x_b ad essere sempre uguale a 10

        x_d = c;
    }
}

... //Classe che chiama il costruttore di D che chiama il costruttore di default
    //di B

```

- Non c'è un costruttore della classe derivata. Come sappiamo in questo caso Java fornisce il costruttore standard che non fa nulla e lascia i campi al loro valore di default. Nel caso di una classe derivata succede questo: quando il costruttore di default della classe derivata viene invocato, quest'ultimo invoca a sua volta il costruttore di default della classe base (anche qui, se questo è inibito o è stato ridefinito private si genererà un errore).

12.6 Costrutto this():

parola chiave accostabile a super(). this permetteva di fare riferimento esplicitamente a campi funzione o dati che stanno nella nostra classe, ma adesso vedremo che permette anche di chiamare i costruttori della stessa classe in cui viene chiamato this(). Come super anche this va sempre e solo come prima istruzione all'interno di un costruttore. Ma perché dovremmo chiamare un costruttore della stessa classe? Si fa quando abbiamo costruttori progressivi nella stessa classe, nel senso che abbiamo un costruttore che specifica un po più di cose rispetto ad un altro; this() in questo caso ci permette di riusare codice già scritto senza riscriverlo. Vediamo un esempio:

```
public class Persona {

    //campi dati (variabili di istanza)
    private String nome;
    private String residenza;

    //costruttori
    public Persona() {
        this("Mario Rossi"); //questo costruttore chiama il secondo costruttore
    }

    public Persona(String n) {
        this(n, null); //questo costruttore chiama il terzo costruttore
    }

    public Persona(String n, String r) {
        nome = n;
        residenza = r;
    }
}
```

Nel costruttore o usiamo super o usiamo this, non possiamo fare tutti e due.

13 Overriding:

l'overriding permette di avere, in una classe derivata, un metodo con stesso nome, stesso tipo di ritorno e stessi parametri in input, di un metodo della classe base (può cambiare solo il modificatore di visibilità, quindi dire stessa segnatura, come scritto sulle slide, è impreciso). Quindi effettivamente permette di sovrascrivere (overriding appunto) un metodo della classe base. Se c'è overriding, il metodo base diventa inutilizzabile per la derivata, a meno di utilizzare super.metodo(). Facciamo un esempio su quando conviene fare overriding: abbiamo un'app per una catena di supermercati, sulla classe base persona c'è una politica scontistica espressa da getDiscount(), poi definiamo la classe studente che estende persona. Per gli studenti la catena decide che le politiche degli sconti sono completamente a parte. A quel punto getDiscount() definita per la classe Persona non va bene per gli studenti e bisogna ridefinirlo.

Nota: Attenzione a non confondere overriding (sovrascrizione) e overloading (sovraffunzione). Comunque si può fare overloading nella classe derivata di un metodo ereditato dalla classe base, cambiando i parametri in input. In questo modo chi crea le istanze della classe derivata può sia chiamare il metodo della classe base che quello overridato della classe derivata, poiché entrambi sono presenti nella classe derivata, uno ereditato (quindi non riscritto esplicitamente) e uno scritto esplicitamente. Questo vedremo non sarà vero quando chi crea le istanze della classe derivata chiamerà un metodo overridato e non overridato.

Se si fa overriding di un metodo, e si invoca la funzione ridefinita, indipendentemente da come ci riferiamo all'istanza (potremmo ad esempio farlo con una istanza della classe base), verrà chiamato il metodo nuovo (questo è un risultato del late binding, v. dopo). Late Binding: **il CORPO della funzione** chiamata non è deciso a tempo di compilazione, ma a tempo di esecuzione. Java a esecuzione dirà:

questo metodo f è chiamato su un riferimento di tipo B, fammi però controllare se sta variabile di tipo B è di qualche tipo più specifico. Cioè poiché l'oggetto è della classe derivata, anche se chiamo il metodo su un'istanza b della base, a *run-time java si accorge del tipo di dato effettivo che b contiene* e chiama il metodo nuovo. Vediamo un esempio per chiarire meglio:

```

public class B {

    public void f(int i) {
        System.out.println(i*i);
    }

}

public class D extends B {

    public void f(String s) { //OVERLOADING: cambiano i parametri in input
        System.out.println(s);
    }

    public void f(int n) { //OVERRIDING: non cambia nulla (sarebbe potuto cambiare public)
        System.out.println(n*n*n);
    }

}

public class Esempio1 {
    public static void main(String[] args) {
        B b = new B();
        b.f(5);           //stampa 25: viene invocato il metodo originale su B usando un'istanza
                          di B.
        D d = new D();
        d.f("ciao");   //stampa ciao: viene invocato il metodo overloaded di D
    }
}

public class Esempio2 {
    public static void main(String[] args) {
        D d = new D();
        d.f(10);         //stampa 1000: viene invocato il metodo di D usando un'istanza di D.
        B b = d;
        b.f(10);         //cosa stampa 100 o 1000? Stampa 1000: viene invocato il metodo di D a
                          prescindere da come ci riferiamo all'istanza, perche' il late binding fa
                          controllare a Java solo a tempo di esecuzione quale funzione deve attivare, e a
                          tempo di esecuzione b è un riferimento a un'istanza di D. Quindi viene invocato
                          il metodo nuovo.
    }
}

```

Facciamo alcune osservazioni:

- Eventuali clienti della classe D non potranno più invocare B.f(int), ma solo D.f(int). Poiché anche se D ha ereditato B.f(int), poi l'ha reso inutilizzabile overridandolo con D.f(int), quindi non è come il caso dell'overloading fatto prima!
- D invece potrà ancora evocare B.f(int) utilizzando super.f().

Nota, annotazione @Override: Aiuta a capire se sto facendo errori nel programma: l'IDE ti dice se lo stai facendo davvero overriding. Facciamo un esempio:

```

public class B {

    public void f(int i) {
        System.out.println(i*i);
    }

}

```

```
public class D extends B {  
  
    @Override  
    public void f(int i) {  
        System.out.println(i*i*i)  
  
    }  
}
```

Se io al posto di quest'ultima classe avessi scritto:

```
public class D extends B {  
  
    @Override  
    public void f(long i) {  
        System.out.println(i*i*i)  
  
    }  
}
```

`@Override` sarebbe stato tratteggiato in rosso, per indicarci che quello sotto non era un ovverride ma solo un overload. Quindi l'IDE ci segnala quando facciamo errori di questo tipo.

14 Late Binding:

Innanzitutto, cos'è un binding? Si tratta del collegamento tra una chiamata di funzione e il posto della memoria dove si trova il codice che esegue quella chiamata. Java questo collegamento (linking, o binding appunto) non lo fa a tempo di compilazione (a tempo di compilazione viene visto solo sintatticamente che il metodo può essere chiamato, cioè si controlla solo l'esistenza di una funzione con la segnatura richiesta), ma a tempo di esecuzione: solo a run-time si crea il collegamento tra chiamata di funzione e corpo della funzione. Questo viene chiamato Late Binding, contrapposto all'Early Binding. Il collegamento fatto a run-time è stabilito in base alla classe dell'oggetto di invocazione, e non in base al tipo del riferimento utilizzato, come abbiamo già visto in un precedente esempio. Cioè il Late Binding si basa sul fatto che il collegamento sia indipendente dal tipo di riferimento utilizzato per riferirsi all'oggetto.

Facciamo un esempio:

```
public class B {  
  
    protected int id;  
  
    public B(int i) { id = i; }  
  
    public boolean get() { return id < 0; }  
}  
  
public class D extends B {  
  
    protected char ch;  
  
    public D(int i, char c) {  
  
        super(i);  
  
        ch = c;  
    }  
  
    public boolean get() { return ch != 'a'; }  
}
```

```

public class LateBinding {

    public static void main(String[] args) {

        D d = new D(1, 'b');

        B b = d;

        System.out.println(b.get());

        System.out.println(d.get());

    }

}

```

Abbiamo una classe B con un metodo get(), e una classe D che estende B e che a sua volta definisce get(): questo fa overriding sul primo get(). Se creiamo un'istanza di D, a prescindere dal tipo di riferimento a cui lo associamo (può essere sia B b sia D d), chiameremo sempre il nuovo get(). Vediamo la classe col main:

- b e d puntano allo stesso oggetto in memoria
- viene chiamato println su b.get(). Questo ritorna true, perché la JVM associa a tempo di esecuzione (Late Binding appunto) il metodo get() da utilizzare alla segnatura, mentre a tempo di compilazione si è solo assicurata che il metodo get() fosse sintatticamente corretto. Infatti, a tempo di esecuzione Java si accorge che b si riferisce a un'istanza di D, e quindi attiva il metodo della classe D, che ritorna true.
- viene chiamato println su d.get(). Questo ritorna true, perché d è un riferimento a un'istanza di D, quindi a tempo di esecuzione Java associa la segnatura al metodo di D e lo attiva, ritornando true.

Quindi vediamo che indipendentemente dal tipo del riferimento utilizzato, si usa il get() overrided.

Note sull'overriding:

- Anche se la segnatura deve rimanere uguale, la funzione overrided può avere un livello d'accesso diverso, ma solo se *allarga* la visibilità. Questo si può fare perché il modificatore di visibilità non fa parte della segnatura. Se provo a modificare il livello d'accesso ma restringendolo ottengo un errore.
- Si può bloccare l'overriding di una funzione qualificandola final. Cioè final applicato a un metodo impedisce di creare funzioni overrided di quel metodo (ma non blocca l'overloading). Final può essere applicato anche a variabili: una volta inizializzate, queste non potranno essere modificate. final infine può essere applicato anche alle classi: una classe final NON PUÒ avere classi derivate. In altri linguaggi final viene chiamato const. La parola chiave final applicata alle variabili è utilizzata anche per andare a definire delle costanti di classe nel programma, ad esempio:

```
public static final int COSTANTEINTERA = 54;
```

Sarebbe buona prassi di programmazione mettere tutto in maiuscolo il nome di queste costanti (molti IDE segnalano un warning se non lo si fa, ad esempio NetBeans)

Nota, refactoring: Altra cosa da imparare negli IDE è il refactoring: in un progetto grosso se cambi nome ad una variabile ce metti anni a ricambiare tutto, invece gli IDE hanno l'opzione refactor, che rinomina tutti i possibili riferimenti a quella variabile in tutto il programma (spesso con la possibilità di farlo anche all'interno dei commenti, ma attenzione con le variabili che si chiamano con una lettera sola in tal caso!). Su NetBeans, vai sulla var, o sul metodo, o sulla classe che vuoi rinominare e col tasto destro fai refactor, poi rename e ti rinomina tutto.

14.1 L'overriding di campi dati non esiste!

Se nella derivata definisci un campo dati con stesso nome di campo dati della base quello della derivata nasconde quello base, ma quest'ultimo rimane accessibile utilizzando riferimenti del tipo base. Quindi **non ci sono Overriding e Late Binding sulle variabili, ma solo sui metodi!** Quindi mentre nel caso dei campi funziona il tipo del riferimento con il quale mi riferisco all'istanza non influenza la scelta del corpo del metodo, nel caso dei campi dati il tipo del riferimento influenza a cosa accedo. Facciamo un esempio:

```
public class B { int i; }

public class D extends B {

    char i;

    void stampa() {

        System.out.println(i);

        System.out.println(super.i);

    }
}

public class SovrascritturaCampiDati {

    public static void main(String[] args) {

        D d = new D();

        d.i = 'f';

        ((B)d).i = 9;

        d.stampa();
    }
}
```

Ok, abbiamo una classe B che ha un campo dati int i, e una classe D che estende B che ha un campo dati char i, e un metodo stampa() che:

- con `System.out.println(i);` stampa ciò che c'è dentro char i, cioè la sua variabile peculiare
- con `System.out.println(super.i)` stampa ciò che c'è dentro int i, cioè la variabile della superclasse.

E ok, anche nel caso di overriding possiamo usare super per accedere alla variabile della superclasse, non è una novità. Ma nel main andiamo a creare istanza di D, e su questa istanza chiamiamo la variabile i, a cui associamo il valore 'f'. Quale i è stata chiamata? In questo caso è stata chiamata la d della classe D, perché d è istanza di questa. Dopo però scriviamo:

```
((B)d).i = 9;
```

Quindi utilizziamo d, ma castandolo a tipo B, e ci invochiamo sopra i, che modifichiamo col valore 9. Quale i è stata chiamata? Proprio perché non c'è overriding, *il fatto che io acceda a i con una variabile di tipo B condiziona la scelta del corpo del metodo*: viene modificata la i della base, perché ho castato d al tipo della base. Se invece avessimo avuto overriding e late binding allora avremmo acceduto a i della classe D perché, a prescindere dal tipo di d, la JVM a run-time si sarebbe accorta che in d c'è un'istanza del tipo D. In conclusione: dato che non c'è overriding per accedere alla var della base basta cambiare il tipo del riferimento, mentre se c'è overriding questo è inutile perché c'è late binding a tempo di esecuzione quindi Java vede direttamente l'istanza dentro il riferimento, senza considerare il tipo di quest'ultimo.

15 Classi Astratte:

Sono classi parzialmente definite. Esse non possono avere istanze proprie, e quindi vengono utilizzate per derivare sottoclassi. Nelle classi astratte solitamente (posso anche avere metodi definiti completamente) si hanno funzioni astratte, cioè dichiarate (segnatura presente) ma non definite (corpo non presente). Le classi derivate, dette concrete, avranno poi il compito di fare overriding di queste funzioni astratte. Perché fare questo? Spieghiamolo con un esempio: ho una classe Figura e ci voglio mettere il metodo area. Questo non posso implementarlo a prescindere dalla figura particolare: il calcolo dell'area è diverso in base alla figura, quindi non posso implementare il corpo a priori. Posso avere un'insieme di metodi definiti a livello classe Figura (quindi sia segnatura sia corpo nella classe astratta) e il resto lo definiscono le classi derivate di Figura. Se provo a istanziare una classe astratta il compilatore mi blocca. Un esempio di funzione astratta:

```
abstract public double getPerimetro();
```

Quindi utilizzato l'aggettivo abstract, e non hanno graffe, quindi niente corpo: c'è solo la segnatura.

Esempio di classe astratta Figura e sue classi derivate concrete:

```
//File figure/Figura.java

package figure;

abstract public class Figura { //classe astratta

    private String colore;

    public Figura(String c) {

        colore = c;

    }

    public Figura() {

        this("bianco");

    }

    public String getColore() {

        return colore;

    }

    abstract public double getPerimetro();

    abstract public double getArea();

}
```

```
//File figure/Cerchio.java

package figure;

public class Cerchio extends Figura {

    private double raggio;

    public Cerchio(String c,double r) {
```

```

super(c);

raggio = r;

}

public Cerchio(double r) {

//super();

raggio = r;

}

public double getPerimetro() {

return 2*Math.PI*raggio;

}

public double getArea() {

return Math.PI*raggio*raggio;

}

}
-----
```

```

//File figure/Rettangolo.java

package figure;

public class Rettangolo extends Figura {

private double base;

private double altezza;

public Rettangolo(String c,double b, double a) {

super(c);

base = b;

altezza = a;

}

public Rettangolo(double b, double a) {

//super();

base = b;

altezza = a;

}

public double getPerimetro() {

return base+altezza+base+altezza;
```

```

    }

    public double getArea() {

        return base*altezza;
    }

}

-----



//File figure/Triangolo.java

package figure;

public class Triangolo extends Figura {

    private double l1;
    private double l2;
    private double l3;

    public Triangolo(String c,double l1, double l2, double l3) {

        super(c);

        this.l1 = l1;

        this.l2 = l2;

        this.l3 = l3;
    }

    public Triangolo(double l1, double l2, double l3) {

        //super();

        this.l1 = l1;

        this.l2 = l2;

        this.l3 = l3;
    }

    public double getPerimetro() {

        return l1+l2+l3;
    }

    public double getArea() { //basato su formula di Erone

        double sp = getPerimetro()/2;

        return Math.sqrt(sp*(sp-l1)*(sp-l2)*(sp-l3));
    }
}

```

}

Osservazioni:

- Math è il package per funzioni e variabili matematiche, da cui prendiamo PI. Vediamo che PI è tutto in maiuscolo, proprio perché è una costante **public static final**, come prima abbiamo spiegato.
- Dal punto di vista della pulizia del codice può essere utile usare **@Override** quando si va a implementare una funzione astratta nella classe concreta. Difatti, se scriviamo Cerchio, Rettangolo o Triangolo su NetBeans, l'IDE ci darà la possibilità (cliccando sulla lampadina) di implementare in automatico i metodi astratti, scrivendoci sopra proprio **@Override**. Nel corpo dei metodi metterà **throw new**, che serve per generare eccezioni, cioè errori.
- Vediamo che è buona norma fare la seguente cosa: nella classe base abbiamo due costruttori, uno senza parametri e uno con il parametro per il colore. In ogni classe derivata abbiamo due costruttori, uno senza parametri della base (cioè senza colore) ma con solo i suoi due parametri peculiari, e uno con il parametro base per il colore e poi i due suoi parametri peculiari.

15.1 Cliente di Classe Astratta e Polimorfismo:

Scrivere una funzione cliente che dato un array di figure, restituisca la somma dei perimetri.

```
//File servizifigure/ServiziFigure.java

package servizifigure;

import figure.*;

public class ServiziFigure {

    public static double perimetroComplessivo(Figura[] a) {

        double ris = 0;

        for (int i = 0; i < a.length; i++)

            ris = ris + a[i].getPerimetro();

        return ris;
    }
}
```

```
//File Main.java

import figure.*;
import servizifigure.*;

public class Main {

    public static void main(String[] args) {

        Figura r = new Rettangolo(10,10);

        Figura t = new Triangolo(10,10,10);

        Figura c = new Cerchio(10);
```

```

Figura[] a = new Figura[3];

a[0]=r;

a[1]=t;

a[2]=c;

System.out.println(ServiziFigure.perimetroComplessivo(a));

}

```

Osservazioni:

- Figura[] a è array di figure. Vediamo questa volta non usiamo il metodo length(), come abbiamo fatto in passato sulle stringhe, ma length, che è una proprietà degli array.
- Potevamo anche scrivere questo per inizializzare l'array:

```
Figura[] a = {new Rettangolo(10,10), new Triangolo(10,10,10), new Cerchio(10)};
```

- vediamo che il riferimento allo specifico oggetto viene tenuto all'interno di var che sono di tipo figura, che è una classe astratta. Quindi la classe astatta non posso istanziarla ma posso farle tenere istanze di sottoclassi concrete
- quando l'array viene passato al perimetroComplessivo, grazie al late binding ogni chiamata di getPerimetro() chiama il metodo specifico associato alla istanza specifica. Questo significa che perimetroComplessivo è **una funzione polimorfa**, poiché a seconda del contesto adotta una versione di un metodo o un'altra. Il *polimorfismo* mi permette di scrivere codice pulito, facilitando leggibilità, riuso ed estendibilità, **astraendo il concetto di figura**.

La cosa importante quindi è che mi basta scrivere perimetroComplessivo e basta per tutte le sottoclassi che vogliamo, grazie al late binding. Perché ho le varie classi che hanno implementato la getPerimetro() in maniera specifica, il late binding decide quale specifica versione va chiamata. **Per supportare il polimorfismo l'overriding e il latebinding sono fondamentali**, come abbiamo capito.

Part V

Polimorfismo:

16 Interfacce Java:

Accanto alle classi ci sono le interfacce. Hanno la stessa sintassi della classe. Dentro l'interfaccia si mettono solo dichiarazioni di segnatura (non corpo) di funzioni, implicitamente pubbliche; quindi non possono essere presenti campi dati (eccetto costanti statiche). Si tratta quindi di una collezione di funzionalità: le classi che implementano l'interfaccia hanno delle funzionalità comuni, appunto le funzionalità dell'interfaccia. *Implemente un'interfaccia* significa definire il corpo dei metodi dell'interfaccia. Vediamo un esempio:

```
public interface Confrontabile { //questa e' l'interfaccia

    //funzioni di cui definire il corpo
    boolean maggiore(Confrontabile c);
    boolean paritetico(Confrontabile c);
}
```

Notare come non abbiamo utilizzato il modificatore di visibilità public. Questo perché le funzioni di un interfaccia sono implicitamente public. Una classe che implementa un'interfaccia utilizza la parola chiave "implements". Ad esempio, queste sono due classi che implementano l'interfaccia Confrontabile:

- Classe Edificio:

```
public class Edificio implements Confrontabile {

    //campi dati
    protected int altezza;

    //costruttori
    public Edificio(int a) { altezza = a; }

    //campi funzioni
    public boolean maggiore(Confrontabile e) {
        if (e != null && e instanceof Edificio) //ignorare per ora
            return altezza > ((Edificio)e).altezza;
        else return false;
    }

    public boolean paritetico(Confrontabile e) {
        if (e != null && e instanceof Edificio) //ignorare per ora
            return altezza == ((Edificio)e).altezza; //il primo
            //altezza dovrebbe essere this.altezza, cioe'
            //relativo all'istanza su cui e' stato chiamato
            //paritetico, mentre il secondo e' relativo all'eta
            //dell'oggetto passato come parametro
        else return false;
    }
}
```

Concentriamoci sulla funzione maggiore (analoghi discorsi valgono per paritetico):

```
public boolean maggiore(Confrontabile e) {
    if (e != null && e instanceof Edificio) //ignorare per ora
        return altezza > ((Edificio)e).altezza;
    else return false;
}
```

Si prende quindi la funzione che era stata definita solo con la segnatura e la si definisce con

il corpo. Cioè si prende il concetto astratto di maggiore e si spiega che cosa vuol dire essere maggiore per un edificio. Essere maggiore per un edificio significa essere più alti. Comunque il codice fa la seguente cosa: prende in input un Confrontabile, controlla che non sia nullo e che sia istanza di Edificio (perché se non lo è che fai confronti qualcosa che non è un Edificio con un Edificio?) e dopodiché fa il confronto, facendo il casting a Edificio perché effettivamente in input abbiamo preso un tipo Confrontabile, e quindi anche se è istanza di Edificio va castato. Fatto il casting, va a prendere il campo altezza. Comunque dopo il comando `instanceof` lo spieghiamo bene.

- Classe Persona:

```

public class Edificio implements Confrontabile {

    //campi dati
    protected int eta;

    //costruttori
    public Persona(int e) { eta = e; }

    //campi funzione

    public int getEta() {return eta; }

    public boolean maggiore(Confrontabile p) {
        if (p != null && p instanceof Persona) //ignorare per ora
            return eta > ((Persona)p).eta;
        else return false;
    }

    public boolean paritetico(Confrontabile p) {
        if (p != null && p instanceof Persona) //ignorare per ora
            return eta == ((Persona)p).eta; //il primo eta dovrebbe
                                             //essere this.eta, cioe' relativo all'eta
                                             //dell'istanza su cui viene chiamato il metodo. Dopo
                                             //vedi c'e' un esercizio con c1.maggiore(c2),
                                             //this.eta e' relativo a c1, mentre il secondo eta e'
                                             //relativo a c2
        else return false;
    }
}

```

Discorso analogo ai metodi maggiore e paritetico di Edificio vale per quelli di Persona.

Bene, ora abbiamo le classi che implementano l'interfaccia e definiscono i suoi metodi. Adesso costruiamo un Servizio:

```

public class ServiziConfrontabile {

    static public Confrontabile MaggioreTraTre(Confrontabile c1, Confrontabile
                                                c2, Confrontabile c3) {
        if ((c1.maggiore(c2) || c1.paritetico(c2)) && (c1.maggiore(c3) ||
                                                       c1.paritetico(c3)))
            return c1;
        else if ((c2.maggiore(c1) || c2.paritetico(c1)) && (c2.maggiore(c3) ||
                                                               c1.paritetico(c3)))
            return c2;
        else return c3;
    }
}

```

Che fa sto servizio? Prende tre confrontabili e restituisce il confrontabile maggiore tra i tre. Più di che cosa fa ci interessa cosa chiama:

```
if ((c1.maggiore(c2) || c1.paritetico(c2)) && (c1.maggiore(c3) || c1.paritetico(c3)))
```

Vediamo se su c1, c2 e c3, che sono Confrontabili, chiama le funzioni maggiore e paritetico, che, proprio in Confrontabile, sono solo definite con la segnatura ma non hanno un corpo definito! Questo Servizio, può essere utilizzato da ogni Classe che implementa confrontabile! Cioè Edificio e Persona, che implementano Confrontabile, avendo un corpo definito e specifico per i metodi maggiore e paritario, potranno usare questo Servizio. Quindi il Servizio è definito a prescindere dalla classe che lo utilizzerà, e a prescindere da ciò che significa concretamente, per quella classe specifica che utilizzerà il servizio, maggiore o paritetico. Adesso facciamo una prova, creiamo un Main e nel main facciamo usare il Servizio alle classi Persona ed Edificio, così da provare ciò che abbiamo appena detto:

```
public class Main{

    public static void main(String[] args) {

        Persona p1 = new Persona(30); //inizializziamo tre persone
        Persona p2 = new Persona(35);
        Persona p3 = new Persona(32);

        Edificio e1 = new Edificio(12); //inizializziamo tre edifici
        Edificio e2 = new Edificio(5);
        Edificio e3 = new Edificio(100);

        Persona pp = (Persona)ServiziConfrontabile.MaggioreTraTre(p1,p2,p3);
        //confrontiamo le tre persone

        Edificio ee = (Edificio)ServiziConfrontabile.MaggioreTraTre(e1,e2,e3);
        //confrontiamo i tre edifici
    }
}
```

Vediamo che nelle ultime due assegnazioni dobbiamo castare il risultato di MaggioreTraTre al tipo giusto per inserirlo nella variabile, perché? Perché MaggioreTraTre, funzione di ServiziConfrontabile, ritorna un confrontabile (infatti fa return c1, c2 o c3, che sono i parametri Confrontabile dati in input).

Tutto questo discorso, relativo alle interfacce, si regge ovviamente su Polimorfismo e Late Binding. Senza di questi non possiamo costruire queste funzioni indipendenti dal particolare codice che poi implementiamo per usarle nel contesto di una classe specifica, quindi non potremmo scrivere il servizio:

```
public class ServiziConfrontabile {

    static public Confrontabile MaggioreTraTre(Confrontabile c1, Confrontabile
        c2, Confrontabile c3) {
        if ((c1.maggiore(c2) || c1.paritetico(c2)) && (c1.maggiore(c3) ||
            c1.paritetico(c3)))
            return c1;
        else if ((c2.maggiore(c1) || c2.paritetico(c1)) && (c2.maggiore(c3)
            || c1.paritetico(c3)))
            return c2;
        else return c3;
    }
}
```

Perché non potremmo fare ad esempio `c1.maggiore(c2)` essendo c1 un confrontabile e non avendo un corpo definito per il metodo maggiore. Allora il **Late Binding** ci permette aspettare il tempo di esecuzione per decidere il corpo della funzione maggiore, in modo che Java si accorga che in realtà c1 non è solo confrontabile ma è specificatamente una Persona, o un Edificio, e così sappia perfettamente qual è il corpo della funzione maggiore da eseguire. Questo rende maggiore() una **Funzione Polimorfa**.

Differenze tra interfacce e classi astratte: una classe astratta è una classe, cioè una astrazione di un insieme di oggetti simili (cioè gli oggetti delle sue sottoclassi), mentre una interfaccia è invece una astrazione di un insieme di funzionalità, quindi solo dei campi funzione. Ma perché ciò che ho fatto con l'interfaccia non posso farlo con una classe astratta? Si può dire che Persona e Edifici siano figlie della stessa cosa, che hanno qualcosa in comune? No! Quindi non ci può essere una classe astratta che le accomuna. Invece si può giustamente dire che hanno delle funzionalità in comune, cioè un'interfaccia. Ecco perché interfaccia e non classe astratta. *Un interfaccia non vuole rappresentare oggetti simili, mentre una classe astratta sì.* Quindi Confrontabile non poteva essere pensata come classe astratta, cioè come astrazione delle Persone e degli Edifici: è una forzatura, poiché Confrontabile è un'astrazione solo delle funzionalità maggiore e paritetico che poi Persona ed Edificio definiscono specificatamente, quindi è un interfaccia. Altre differenze tra classe astratta e interfacce:

- Le classi astratte vengono estese, mentre le interfacce vengono implementate.
- Mentre di una classe se ne può estendere solo una di interfacce ne puoi implementare più di una, quante te ne pare.

Altre caratteristiche delle Interfacce Java:

- Un classe può implementare più di una interfaccia. C'è quindi una sorta di Ereditarietà Multipla²² (che per le classi non c'era!). Cioè posso avere una classe che fa implements di più di una interfaccia (questa cosa non è possibile con le classi astratte, dato che una classe può estendere solo un'altra classe al massimo). Facciamo un esempio:

```

public interface I { void g(); } //prima interfaccia con una funzione di cui
                                bisognerà definire il corpo

public interface J { void h(); } //seconda interfaccia con una funzione di cui
                                bisognerà definire il corpo

public class C implements I,J { //classe che implementa entrambe le
                                interfacce

    void g() {...} //qui il corpo della funzione della prima interfaccia
                    viene definito
    void h() {...} //qui il corpo della funzione della seconda
                    interfaccia viene definito
}

```

- Un'interfaccia può essere derivata da un'altra interfaccia. Cioè posso avere una interfaccia che è figlia di un'altra interfaccia ed eredita le segnature delle funzioni dell'interfaccia madre. Facciamo un esempio:

```

public interface I { void g(); } //interfaccia "madre"

public interface J extends I { void h(); } //interfaccia derivata, eredita
                                         g() da I

public class C implements J { //classe che implementa la interfaccia
                                derivata, andando a definire il corpo di tutte e due le funzioni, sia
                                della funzione peculiare della derivata sia della funzione che
                                quest'ultima ha ereditato dalla interfaccia "madre"

    void g() {...}
    void h() {...}
}

```

²²Una sorta perché la classe che implementa non eredita nulla, solo l'obbligo di definire il corpo delle funzioni dell'interfaccia.

- Java supporta *Ereditarietà multipla per le interfacce* (sempre non per le classi). Cioè oltre a poter fare *implements* di più interfacce su una classe, si può fare *extends* di più interfacce su una stessa interfaccia. Quindi una interfaccia derivata può avere più di una interfaccia madre, cosa assolutamente negata alle classi. Facciamo un esempio:
-

```
public interface I {void g();}

public interface J {void h();}

public interface M extends I,J {
    void k();
}

public class C implements M {
    void g(){...} void h(){...} void k(){...}
}
```

17 La Classe Object:

Tutte le classi Java sono IMPLICITAMENTE derivate dalla classe Object, e quindi *ne ereditano le funzioni*. Questo significa anche che *tutti gli oggetti, di qualunque classe, sono istanze di Object* e su di essi possono sempre essere invocati i metodi della classe Object. Le funzioni di Object sono:

- public String *toString()*. Questa funzione associa all'oggetto di invocazione una stringa stampabile. Il *toString()* di Object scrive una stringa pseudocasuale che non è specificato come è ottenuta, ogni implementazione della JVM decide come farla, ha sempre @ stringa alfanumerica e poi il tipo. Si può fare l'*overriding* di questa funzione per stampare una rappresentazione testuale dell'oggetto. Quindi in pratica la rifacciamo noi. Facciamo un esempio:
-

```
public class B {

    private int i;

    public B(int x) { i = x; }

    @Override
    public String toString() { return "i: " + i; }

}

public class EsempioToString {

    public static void main(String[] args) {

        B b = new B(5);
        System.out.println(b);

    }

}
```

Nel main c'è *println(b)*, quando gli passo un object (quindi è *println versione su oggetti*, cioè su object) chiama il *toString()*. Se è definito chiama quello, se no chiama quello di Object. Il seguente codice stampa "i: 5". Notare che se non avessimo ridefinito *toString()* facendo overriding, il *toString()* della classe Object avrebbe stampato "B@601bb1". Vediamo che possiamo ovviamente fare overriding dell'overriding, e poi riusare il *toString()* della superclasse. Facciamo un esempio:

```

public class B {

    private int i;

    public B(int x) { i = x; }

    @Override
    public String toString() { return "i: " + i; } //primo overriding:
                                                    sovrascrive la funzione toString() ereditata da Object

}

public class D extends B {

    private int j;

    public D(int x, int y) {

        super(x);
        j = y;
    }

    @Override
    public String toString() { //secondo overriding: sovrascrive la
                            funzione toString() ereditata da B.
                            return super.toString() + " j: " + j; //all'interno della
                            funzione che sovrascrive toString() della superclasse
                            quest'ultima viene usata
    }

}

public class EsempioToStringDerivata {

    public static void main(String[] args) {

        D d = new D(5,10);
        System.out.println(d);
    }

}

```

Vediamo che invocando la terza classe, viene stampato "i: 5 j:10", perché vengono attivate le funzioni `toString()` sia di `B` che di `D`. Comunque in realtà: `toString()` non viene usato spesso in programmazione, ma noi lo useremo per fare il debugging dei nostri programmi.

- `public Final Class getClass():` prende l'oggetto di invocazione e restituisce la classe a cui appartiene. **Se non viene invocato su un'istanza specifica restituisce un riferimento alla classe in cui è stato invocato, come se avessimo scritto `this.getClass()`.**
- `public boolean equals(Object o):` prende un oggetto e verifica che sia uguale a un altro oggetto
- `protected Object clone():` prende un oggetto e ne fa una copia
- `public int hashCode():` `hashCode` e `equals` vanno insieme. `hashCode` prende un oggetto e restituisce un codice hash, un codice numerico. Se due oggetti sono uguali allora il loro `hashCode` è lo stesso, il viceversa non è vero: due oggetti con lo stesso `hashCode` non sono per forza uguali.
- altre...

Per le ultime tre funzioni: equals, clone e hashCode, ci dedicheremo un capitolo intero.

18 La Classe Class:

Un'altra classe predefinita, si tratta di una **metaclasse** (??????????). Ricordate che quando compiliamo quello che viene generato è un file con estensione .class? Per ogni classe o interfaccia del programma, nella JVM c'è un oggetto che denota lo denota, e questo oggetto è di tipo Class. Questi oggetti che denotano le classi e le interfacce del programma possono essere nominati:

- Tramite letterali di tipo Class: C.class. Quindi posso chiamare così l'istanza associata alla mia classe, oppure:
- Tramite riferimenti di tipo class: Class c = Cioè posso proprio creare dei riferimenti, delle variabili di tipo classe.

Non possiamo fare il new di un oggetto di tipo Class, cioè non possiamo istanziarlo (difatti Class non ha neanche costruttori accessibili ai clienti). Questo perché gli oggetti di tipo Class sono generati in automatico a runtime. La classe Class ha la funzione:

```
boolean isInstance(Object o)
```

Che restituisce true sse o denota un oggetto che è istanza della classe o dell'interfaccia su cui isInstance viene evocato. Ricordiamo che un oggetto è di una classe se è stato generato col new di quella classe o di una classe derivata di quest'ultima. Facciamo un esempio:

```
public class B {}

public class D extends B {}

public class EsempioIsInstance1 {

    public static void main(String[] args) {

        B b1 = new B();
        D d1 = new D();

        System.out.println(B.class.isInstance(b1)); //true: b1 e' istanza di B
        System.out.println(B.class.isInstance(d1)); //true: d1 istanza di D derivata di B
        System.out.println(D.class.isInstance(d1)); //true: d1 istanza di D
        System.out.println(D.class.isInstance(b1)); //false: b1 non e' istanza di D
    }
}
```

Come già detto funziona anche per le interfacce, anche loro hanno la classe Class. Facciamo un esempio:

```
public interface I {}

public class D implements I {}

public class EsempioIsInstance2{

    public static void main(String[] args) {

        I i1 = new D();
        D d1 = new D();

        System.out.println(I.class.isInstance(i1)); //true: i1 contiene istanza di D che
            implements I
        System.out.println(I.class.isInstance(d1)); //true: d1 e' istanza di D che
            implements I
        System.out.println(D.class.isInstance(d1)); //true: d1 e' istanza di D quindi
            appartiene alla classe D
        System.out.println(D.class.isInstance(i1)); //true: i1 contiene istanza di D
            quindi appartiene alla classe D (?)
    }
}
```

```
    }  
}
```

Quindi `isInstance` può essere usata sia per verificare se un oggetto è istanza di una *classe*, sia se un oggetto implementa una *interfaccia*. Oltre a `isInstance`, esiste anche il costrutto `instanceof`, che fa la stessa cosa (ma è costrutto come `j`, `i`, `o ==`). `instanceof` verifica che un oggetto sia istanza di una classe o implementi una interfaccia, quindi fa la stessa cosa di `isInstance`. Cioè queste due espressioni fanno la stessa cosa:

```
C.class.isInstance(c)
```

```
c instanceof C
```

`instanceof` è però meno potente di `isInstance`: `instanceof` non fa uso di un oggetto `Class` ma del nome della classe, che quindi deve essere noto a tempo di compilazione e non a runtime come con `isInstance`. Cioè in

```
c instanceof C
```

`c` per forza deve essere una variabile mentre `C` per forza classe, mentre in

```
b.isInstance(c)
```

abbiamo scritto `b` al posto di `C.class`, cioè abbiamo sostituito il letterale con una variabile di tipo `class` (tramite assegnazione a riferimento `Class c = ...`) e funziona lo stesso. Questo perché a runtime Java si accorge che `b` contiene una variabile di tipo `class`, mentre a tempo di compilazione vede solo il nome `b` e non capisce.

Torniamo a `getClass()`: La funzione `getClass()` di `Object` restituisce il riferimento all'oggetto `Class` che rappresenta la *classe che è stata usata dal new per creare l'oggetto su cui `getClass()` viene invocato*, ciò significa che restituisce la *classe più specifica* di cui l'oggetto è istanza. Facciamo un esempio:

```
d1.getClass() //se d1 = new D(), restituisce D.class, riferimento all'oggetto Class  
che rappresenta la classe D
```

`getClass()` serve per verificare a runtime *di che tipo è l'oggetto di invocazione di `getClass()`, indipendentemente dal riferimento usato per denotarlo*. Vedremo nel prossimo capitolo un uso tipico di questo metodo. Quindi anche qui, l'importante (cioè quello che verrà restituito) è la classe dell'istanza a cui l'oggetto di invocazione si riferisce, non la classe del riferimento che l'oggetto di invocazione utilizza per riferirsi all'istanza.

19 La classe String Tokenizer, le matrici e il ciclo for each:

In python per dividere la stringa rispetto a un carattere preciso si usava `split`, che restituiva un array. In java è diverso: bisogna creare un'istanza della classe `String Tokenizer`. Un'istanza di `String Tokenizer` è un cursore sulla stringa. Ci sono diversi metodi che permettono di navigare i differenti item che troviamo nella stringa tokenizzata, e vanno invocati sul cursore, cioè sull'istanza. Ad esempio:

- `nextToken()`. Sposta il cursore avanti di un token e lo restituisce (non si può tornare indietro)
- `hasMoreTokens`. Booleano che ci dice che esistono altri token in base a dove è il cursore.

Facciamo un esempio (in cui usiamo differenziacicini, ma poco importa), in cui facciamo vedere anche le **matrici** in Java; inoltre facciamo vedere un **modo ulteriore per fare un ciclo**:

```
import java.util.Scanner;  
import java.util.StringTokenizer;  
  
public class MatrixUtils {  
  
    public static void stampaMatrice(double[][] matrice) {
```

```

//qui non utilizziamo il ciclo for ma il for each:
//se abbiamo una collezione di valori (come un array) puo' essere
//utile accedere a questo senza usare le posizioni (gia' lo abbiamo fatto in
//Python:
//for elem in array[]), ma la sintassi in Java e' un po' differente

//il for each non puo' essere sempre utilizzato: se ho bisogno di conoscere
//la posizione specifica (i,j) dove sono non posso usarlo, perche' perdo questa
//informazione.

for (double[] row: matrice) { //iterazione su righe
    //la variabile row ha scope solo in questo blocco
    //lo scope e' l'area del codice in cui posso usare la variabile

    for (double val: row) { //ciclo sulla riga, che e' un array
        //e metto in val il valore
        System.out.print(val + ",\t");
    }
}

}

public static double[][] differenzaVicini(double[][] matrice) {

    double[][] retValue = new double[matrice.length][matrice[0].length];

    for (int i = 0; i < matrice.length; i++) {

        int up = i-1;
        if (i == 0) {
            up = i;
        }

        int dw = i+1;
        if (i == matrice.length-1) {
            dw = i;
        }

        for (int j = 0; j < matrice[i].length; j++) {

            int sx = j-1;
            if (j == 0) {
                sx = j;
            }

            int dx = j+1;
            if (j == matrice[i].length-1) {
                dx = j;
            }

            //calcolo minuendo - sottraendo

            double sottraendo = 0;

            for (int i1 = up; i1 <= dw; i1++) {
                for (int j1 = sx; j1 <= dx; j1++) {
                    if (i1 != i || j1 != j) {
                        sottraendo += matrice[i1][j1];
                    }
                }
            }
        }
    }
}

```

```

        //mettiamo il risultato nella nuova matrice
        retValue[i][j] = matrice[i][j] - sottraendo;

    }

}

return retValue;
}

public static void DifferenziaViciniDaConsole() { //ECCO IL METODO CHE CI INTERESSA

Scanner sc = new Scanner(System.in);

 StringTokenizer st = new StringTokenizer(sc.nextLine(), ",");

int righe = Integer.parseInt(st.nextToken());
int colonne = Integer.parseInt(st.nextToken());

double[][] matrice = new double[righe][colonne];

for (int i = 0; i < righe; i++) {
    StringTokenizer st2 = new StringTokenizer(sc.nextLine(), ",");
    for (int j = 0; j < colonne && st2.hasMoreTokens(); j++) {
        matrice[i][j] = Double.parseDouble(st2.nextToken());
        //analogo a Integer.parseInt
    }
}

double[][] matricenuova = MatrixUtils.differenzaVicini(matrice);

System.out.println("\n\n\n");
MatrixUtils.stampaMatrice(matricenuova);

}

}

-----



public class Main {

public static void main(String[] args) {

MatrixUtils.DifferenziaViciniDaConsole();

}
}

```

20 Uguaglianza tra Oggetti:

Ma che significa che due oggetti sono uguali? Abbiamo due tipi diversi di uguaglianza: uguaglianza superficiale significa che due riferimenti denotano lo stesso oggetto, mentre l'uguaglianza profonda è quando le informazioni contenute nell'oggetto sono uguali (sembra più forte la superficiale, è contortintuitivo, basta che confronto gli indirizzi in memoria per verificare se sono uguali superficialmente, per questo si chiama così; mentre profonda devo confrontare campo per campo; quindi superficiale e profonda dal punto di vista delle operazioni che devo fare).

20.1 Metodo equals():

Restituisce true se l'oggetto di invocazione è uguale all'oggetto che viene passato, quest'ultimo è un'istanza di Object. La sintassi è:

```
public boolean equals(Object o)
```

Facciamo un esempio

```
public class C { ... }

public class main { C c1 = new C(4,5); C c2 = new C(4,5)}
```

La funzione equals, se non ridefinita, fa l'uguaglianza superficiale, come fa il "==" . In questo caso però possiamo verificare che c1 e c2 non sono uguali superficialmente, ma profondamente.

Si fa overriding del metodo equals() per ottenere l'uguaglianza profonda. Per fare ciò seguiamo un *design pattern* (serie di passi, schema di programmazione):

```
//Pattern per l'overriding di equals()

public class B {
    private int x, y;
    ...

    public boolean equals(Object o) {
        if (o != null &&
            getClass().equals(o.getClass())) {
            B b = (B)o;
            //test uguaglianza profonda
            return (x==b.x) && (y==b.y);
        }
        else return false;
    }

    ...
}
```

Osservazioni:

- getClass().equals(o.getClass()). se devo fare l'uguaglianza profonda per forza di cose devono essere della stessa classe. Notiamo che il primo getClass() non ha un'istanza di invocazione, e quindi restituisce un riferimento alla class B, in cui il metodo getClass() è stato invocato.
- B b = (B)o. Adesso o denota un oggetto di tipo B, quindi lo assegno a un riferimento di tipo B per poter accedere ai suoi campi privati. I campi privati infatti sono accessibili anche da un'altra istanza della stessa classe. Non solo l'istanza di quei campi può accedere ai campi privati, qualunque istanza della stessa classe può accederci. Ciò significa che il modificatore d'accesso ragiona a livello di classi.

Nota: Se come campi dati della classe ci sono altre classi (tipo scuola che ha Insegnante come campo dati), devo fare equals overridato su quei campi.

Di Equals NON SI FA MAI overloading: Su equals si fa overriding ma mai non overloading, perché si potrebbero avere risultati controintuitivi. Vediamo perché, facciamo un esempio in cui usiamo il parametro di equals con B e non object:

```
//EqualsOverloaded
public class B {
    private int x, y;
    public B(int a, int b) {
        x = a; y = b;
    }
}
```

```

public boolean equals(B b) {
    //OVERLOADING, NON OVERRIDING
    if (b != null)
        return (b.x == x) && (b.y == y);
    else return false;
}

public class EqualsOverloaded {
    static void stampaUguali(Object o1, Object o2) {
        if (o1.equals(o2)) System.out.println("I DUE OGGETTI SONO UGUALI");
        else System.out.println("I DUE OGGETTI SONO DIVERSI");
    }
    public static void main(String[] args) {
        B b1 = new B(10,20); B b2 = new B(10,20);
        if (b1.equals(b2)) System.out.println("I DUE OGGETTI SONO UGUALI");
        else System.out.println("I DUE OGGETTI SONO DIVERSI");
        stampaUguali(b1, b2);
    }
}

```

StampaUguali chiamerà la funzione più specifica di equals, essendo i parametri Object, chiamerà equals originale di Object, che fa l'uguaglianza superficiale. Ciò vuol dire che nel main la prima stampa dice che sono uguali, la seconda dice che sono diversi. **Quindi fare sempre overriding e non overloading.**

20.2 hashCode e overriding di hashCode:

Metodo hashCode(): è una funzione di hashing, cioè una funzione one way, non invertibile, che mappa i valori di un dominio, solitamente di cardinalità infinita, su un altro dominio, solitamente di cardinalità più piccola, finita. Mappando su un dominio più piccolo per forza di cose ci potranno essere due oggetti iniziali che hanno lo stesso valore nel dominio più piccolo. Anche se è difficile che ci sia una collisione. hashCode() si usa principalmente per implementazione efficiente di dizionari. Le funzioni di hash sono importanti nella crittografia: valori vicini nel dominio sono valori molto lontani nel codominio. Nella crittografia viene utilizzata per la non viabilità dei documenti. Torriamo ad hashCode: hashCode() definisce l'hashCode dell'istanza della classe, utilizzeremo questo metodo per l'esigenza di alcune strutture dati, come gli HashSet.

Overriding di hashCode(): quando si fa l'overriding di equals si deve necessariamente fare anche overriding di hashCode: sintatticamente il compilatore non lo richiede, ma bisogna farlo sennò si presentano condizioni particolari. Questo perché molte collezioni (ad esempio *HashSet*) in java si basano sulla coppia equals hashCode per fare il controllo. Deve quindi valere la seguente implicazione (non quella inversa):

```
o1.equals(o2) IMPLICA -> o1.hashCode() == o2.hashCode()
```

L'implicazione inversa non è vera. Ok ma che fa hashCode()? Restituisce un intero che verrà calcolato usando gli stessi campi usati nella verifica di uguaglianza profonda (v. esempio). Ci sono diversi pattern per fare hashCode(), noi vediamo questo:

```

//Pattern per l'overriding di hashCode()

public class B {
    private int x, y;
    ...

    public boolean equals(Object o) {
        if (o != null &&
            getClass().equals(o.getClass())) {
            B b = (B)o;
            //al posto di questo commento ci va il test d'uguaglianza profonda specifico per il
            //caso
            return (x==b.x) && (y==b.y); //usa i campi x e y
    }
}
```

```

        }
        else return false;
    }

    public int hashCode() {
        return x+37*y; //funzione di hash
                      //basata sui campi x e y
    }
}

```

in cui otteniamo l'hash dell'istanza facendo una combinazione lineare dei campi della classe di quell'istanza. La scelta del coefficiente 37 non è casuale: si utilizzano numeri primi come coefficienti

20.3 Overriding di equals() nelle classi derivate:

Si fa overriding di equals() nelle classi derivate richiamando prima l>equals() della classe base (quindi si favorisce il riuso del codice). Poi per i campi peculiari della classe derivata si verifica l'uguaglianza dei campi a parte. Il design pattern è il seguente, notare che è ricorsivo: se c'è un'altra classe derivata essa chiamerà la superclasse sopra, e così via fino alla base.

```

//Pattern per l'overriding di equals()
//nelle classi derivate

public class D extends B {
    private int z;
    ...

    public boolean equals(Object o) {
        if (super.equals(o)){
            D d = (D)o;
            //al posto di questo commento ci va il test d'uguaglianza profonda dei campi propri
            //della classe specifico per il caso
            return z==d.z;
        }
        else return false;
    }

    ...
}

```

Notare che non si fanno i controlli per verificare che o non sia nullo o che appartenga alla stessa classe dell'istanza della classe perché sono controlli già fatti da super.equals(o). **super.metodo()** chiama il metodo della classe madre su un'istanza della classe madre, quindi è come se super fosse un'istanza della classe madre su cui viene invocato metodo(). Ovviamente se facciamo overriding di equals dobbiamo farlo anche di hashCode():

```

//Pattern per l'overriding di hashCode()
//nelle classi derivate
public class D extends B {
    private int z;
    ...

    public boolean equals(Object o) {
        if (super.equals(o)){ //usa campi della classe base
            D d = (D)o;
            //al posto di questo commento test uguaglianza profonda
            //dei campi propri della classe specifico per il caso
            return z==d.z;    //piu' campi addizionali
        }
        else return false;
    }

    public int hashCode() {
        return
            super.hashCode() + //usa campi della classe base

```

```

    53*z;           //piu' campi addizionali
}
...
}

```

Quindi in pratica prendiamo l'hashCode della classe base e ci sommiamo qualcos'altro: continuiamo la combinazione lineare.

21 Copia di Oggetti:

Anche qui abbiamo copia superficiale e copia profonda. La copia superficiale è la copia del riferimento all'oggetto. La copia profonda crea un clone dell'oggetto originario, quindi i campi sono uguali ma con un diverso identificatore. Vediamo un esempio di *copia superficiale*:

```

public class C {...}

C c1 = new C(4,5);
C c2;
c2 = c1;

```

Invece per la *copia profonda* si usa `clone()`. Questo in `Object` è definito come `protected`, quindi è accessibile solo alle classi derivate e non ai clienti delle classi derivate. Possiamo però fare overriding, mantenendo la stessa segnatura ma cambiando il livello di accesso, allargandolo. Quindi overridandolo lo portiamo a `public`. Si può dire quindi che `clone()` di default è inibito sulle classi, perché è vero che le classi derivate ereditano `clone` ma i clienti di queste classi derivate non possono usarlo per generare i cloni, quindi è inutile. Quindi ecco un nuovo **design pattern** da imparare bene per l'esame.

```

//Pattern per l'overriding di clone()

public class B implements Cloneable {
    private int x, y;
    ...

    public Object clone() {
        try{
            B b = (B)super.clone();
            //al posto di questo commento ci va la copia profonda dei campi dati specifica per
            //il caso
            //includere liste, array, ecc.
            return b;
        } catch(CloneNotSupportedException e) {
            throw new InternalError("...");
        }
    }

    ...
}

```

Tutte le classi che fanno overriding di `clone` devono per forza implementare l'interfaccia `Cloneable`. Questa interfaccia non contiene nulla (*quindi attenzione che il metodo `clone()` non sta in `Cloneable` ma nella classe `Object`*), quest'implementazione è solo uno dei modi per marcare delle classi in qualche maniera. Se una classe implementa un'interfaccia, quella classe diventa riconoscibile, e facendo `InstanceOf` verifichi che la classe ha `Cloneable` sopra. Vediamo che l'overriding di `clone` passa da `protected` a `public`. Rendendo `clone()` `public` lo rendiamo disponibile ai clienti

try { ... } catch: Ci sono delle eccezioni che sono caught ed eccezioni che sono uncaught. Fino ad ora abbiamo visto uncaught, eccezioni che non devo per forza gestire (esempio: null pointer exception). Quelle caught vanno gestite: il metodo `super.clone()` può generare un'eccezione che va per forza gestita ed è `CloneNotSupportedException`. Gestire un'eccezione significa catturarla quell'eccezione e non far fermare il programma per quella cosa, farlo andare avanti con ciò che scrivi nelle parentesi graffe dopo `catch`. Ci possono essere più blocchi `catch`. Nell'esempio sopra

non voglio davvero gestire l'eccezione, semplicemente la rilancio con throw new con InternalError, che fa terminare il programma. Come si indica che un metodo genera delle eccezioni? Dopo la segnatura del metodo si mette `throws`, infatti il `clone()` di default di `Object`, si può vedere dalla Java Doc, è definito così:

```
protected Object clone() throws CloneNotSupportedException
```

Che appunto ci dice che `clone()` genera un'eccezione: `Clone notSupported Exception`.

Cosa fa il codice sopra: Prima cosa che si fa: si chiama il metodo `super.clone()`: possiamo chiamare `clone` perché siamo in una classe derivata della classe `Object`. `super.clone()` fornisce una copia della classe in cui di tutte le var si è fatta **copia superficiale**. Inoltre `super.clone()` restituisce `Object`, quindi va castato, in questo caso a (B). Comunque, se io delle var devo fare copia profonda, dopo aver fatto `super.clone()` devo fare la copia profonda (perché magari abbiamo campi che rappresentano liste, array ecc.) esplicitamente (quindi nel codice quel commento intende che andrebbe fatto là). `super.clone()` è **necessario**, non c'è un altro modo per fare clonazione superficiale dei campi tutta in un colpo (l'altro modo sarebbe creare un'istanza e riempire manualmente uno ad uno ogni campo). Questo è l'unico modo generico di fare questa cosa. Infine restituiamo b.

Implementare Cloneable: Per fare overriding di `clone()` dobbiamo implementare `Cloneable`, in modo da marcare come clonabili gli oggetti della classe che userà `clone()` e che implementa `Cloneable`

21.1 Overriding di `clone()` nelle classi derivate:

Così come nell'`equals` bisognava richiamare `equals()` della classe base si fa lo stesso in `clone()`, facendo poi copia profonda degli altri campi, quelli peculiari della classe derivata. Vediamo il *pattern*:

```
//Pattern per l'overriding di clone()
//nelle classi derivate

public class D extends B {
    private int z;
    ...

    public Object clone() {
        D d = (D)super.clone();
        //al posto di questo commento ci va la copia profonda dei campi dati specifica per
        //il caso
        //incluso liste, array, ecc.
        return d;
    }

    ...
}
```

Anche qui è presente l'aspetto ricorsivo. Perché qui non abbiamo usato `try {...} catch`? Innanzitutto perché l'eccezione del `clone()` di default viene già gestita dal metodo `clone()` di B. In secondo luogo perché quando abbiamo fatto override del `clone()` di default, quello di `Object`, che ricordiamo essere questo:

```
protected Object clone() throws CloneNotSupportedException
```

Abbiamo ottenuto il seguente metodo di B, overridato:

```
public Object clone() {
    try{
        B b = (B)super.clone();
        //al posto di questo commento ci va la copia profonda dei campi dati
        //incluso liste, array, ecc.
        return b;
    } catch(CloneNotSupportedException e) {
```

```

        throw new InternalError("...");
    }
}

```

Che utilizzava try {...} catch per gestire l'eccezione del clone() di default (invocato con super.clone()). Eccezione che lo stesso clone() di default aveva dichiarato usando dopo la sua segnatura la parola chiave **throws**. Ma ora questo metodo overridato non ha dichiarato di generare eccezioni: non ha la parola chiave **throws** dopo la segnatura. Infatti l'eccezione che il clone() overridato lancia è InternalError, che è del tipo InternalServerError, ed è un'eccezione *uncaught*, quindi non necessita di essere gestita e dichiarata con **throws**. Ciò significa che il nuovo clone(), che overrida nella classe derivata il clone() overridato, non deve gestire nessuna eccezione usando try {...} catch. Questa cosa ci dice una cosa importante: **l'overriding è overriding anche al netto delle eccezioni**. Cioè il metodo che overrida non deve avere per forza lo stesso insieme di eccezioni di quello che sovrascrive per essere considerato overriding.

Nota sull'implementazione di Cloneable: Vediamo inoltre che nella classe derivata non abbiamo implementato Cloneable perché basta che la classe base che implementi Cloneable.

22 Oggetti Mutabili e Immutabili:

22.1 Oggetti mutabili:

Il loro stato può cambiare durante l'esecuzione del programma. Hanno metodi che fanno side-effect (v. dopo). *Vengono usati per rappresentare entità*, o valori complessi come **collezioni** (che vedremo più avanti).

22.2 Oggetti immutabili:

Il loro stato non può cambiare durante l'esecuzione del programma. La classe String ha istanze immutabili. Se fai metodi viene restituita una nuova stringa, non viene modificata quella su cui operi. Le funzioni su di loro *non fanno side-effect*. Cos'è il side-effect? Si tratta del caso in cui l'invocazione di un metodo modifica l'oggetto su cui ho invocato il metodo. *Le classi immutabili solo usate per rappresentare valori*. Quindi un valore è colui che risponde alla domanda "Se prendo questa istanza e ci applico questo metodo ottengo la stessa istanza modificata o un'altra istanza" con "ottengo un'altra istanza, perché i valori non sono modificabili". Questo ha senso, se ho il valore 3 e ci sommo 1, sto ottenendo il valore 3 modificato? No! Sto ottenendo il valore 4, un altro valore.

22.3 Equals() e Clone() per oggetti Mutabili e Immutabili:

22.3.1 Oggetti Mutabili:

Entità: Se io sto rappresentando un *entità* l'identificatore (cioè l'indirizzo dell'oggetto) stabilisce l'uguaglianza²³, quindi non va fatto l'overriding di equals(), poiché quello di default fa già l'overriding superficiale che ci serve. Inoltre non vogliamo cloni di entità tipicamente, quindi neanche clone() si overrida, perché già di default ci fa la copia superficiale che ci serve.

Valori Complessi, come le Collezioni: Per quanto riguarda i valori complessi, come le collezioni (es.: liste), l'identificatore non è significativo, e va fatto l'overriding di equals() per realizzare l'*uguaglianza profonda*, che questa volta è importante. Inoltre ha senso anche fare overriding di clone() per permettere la copia profonda dei valori.

22.3.2 Oggetti Immutabili:

In questo caso equals() va ridefinito, poiché gli oggetti immutabili rappresentano valori. Es. Stringhe: se io ho due stringhe uguali, cioè con lo stesso valore, e non ridefinisco equals(), mi dà true solo quando i due riferimenti puntano allo stesso oggetto, e non quando le due stringhe hanno lo stesso valore. Java fa infatti overriding di equals() nella classe string. Clone() invece non

²³un esempio: due Persone, con stesso nome, cognome e anno di nascita, non sono la stessa persona! In questo caso sono la stessa persona solo se sono lo stesso oggetto

viene ridefinito, perché siccome gli oggetti sono immutabili e quindi nessun metodo può fare side-effect sull'istanza ma restituisce una copia modificata dell'istanza, non è mai necessario fare copie dell'istanza²⁴, ma possiamo usare un nuovo riferimento che denoti lo stesso oggetto immutabile, appunto utilizzando il clone() di default che fa copia superficiale.

23 Astrazione di Valore ed Astrazione di Entità in Java:

23.1 Astrazione di Valore:

23.1.1 Astrazione di Valore (Semplice):

ad esempio un NumeroComplesso. Ecco la rappresentazione di tipo di dato astratto NumeroComplesso, indipendente e a priori rispetto a qualsiasi linguaggio di programmazione Osservazioni:

Tipo di dato astratto NumeroComplesso

```

TipoAstratto NumeroComplesso
Domini
    C : dominio dei complessi – dominio di interesse
    R : dominio degli reali
Funzioni
    creaComplesso(R r, R i) → C
        pre: nessuna
        post: RESULT è il numero complesso avente r come parte reale e i come parte
              immaginaria
    reale(C c) → R
        pre: nessuna
        post: RESULT è il valore della parte reale del numero complesso c
    immaginaria(C c) → R
        pre: nessuna
        post: RESULT è il valore della parte immaginaria del numero complesso c
    modulo(C c) → R
        pre: nessuna
        post: RESULT è il modulo del numero complesso c
    fase(C c) → R
        pre: nessuna
        post: RESULT è la fase del numero complesso c
FineTipoAstratto

```

È una astrazione di valore (semplice)

- Il dominio di interesse è *l'insieme degli elementi PROPRI DEL TIPO*, poi ci possono essere, come in questo caso, altri domini che sono insiemi di altri elementi che il tipo usa o di cui è formato.
- La funzione creaComplesso sarebbe il costruttore se fossimo in Java, ma essendo una rappresentazione del tipo astratto non lo è
- Per il fatto che è una rappresentazione astratta, non si parla di OOP. Se volessimo poi implementarlo in OOP, allora NumeroComplesso diventerebbe una classe e queste funzioni diventerebbero i metodi della classe
- Si parla di **astrazione di valore semplice** perché un numero complesso è un valore, quindi un oggetto immutabile, poiché non ha senso modificarlo ma invece è ragionevole pensare di operare su questo creando un nuovo valore. Si parla di valore semplice, per distinguerlo dal valore complesso che è invece mutabile, perché il numero complesso non è un valore formato da altri valori. Comunque se volessimo potremmo implementare questa astrazione con side-effect, quindi più che un valore il numero complesso si potrebbe considerare un'entità: magari stai facendo un app che ha come "personaggi principali" i numeri, tipo che so un app sulla Cabala; allora in questo caso è ragionevole pensare i numeri come entità e potrebbe avere senso modificarli per qualche ragione.

²⁴esempio: se ho il valore 4, perché mai dovrei fare una copia del valore 4 e avere due valori 4 in memoria? Non mi serve a nulla con i valori la copia profonda!

Questo per i dati immutabili, passiamo alle due tipologie di dati mutabili, valori complessi e entità:

23.1.2 Astrazione di Valore Complesso (collezioni):

Facciamo un esempio di collezione, cioè di valore complesso (che si distingue dal valore semplice perché è mutabile, diversamente dal valore semplice). Nota che qui insieme Vuoto() sarebbe il

Tipo di dato astratto Insieme(T)

TipoAstratto Insieme(T)

Domini

Ins : dominio di interesse

T : dominio degli elementi dell'insieme

Funzioni

insiemeVuoto() \rightarrow *Ins*

pre: nessuna

post: RESULT è l'insieme vuoto

estVuoto(Ins i) \rightarrow Boolean

pre: nessuna

post: RESULT è true se *i* è il valore corrispondente all'insieme vuoto, false altrimenti

inserisci(Ins i, T e) \rightarrow *Ins*

pre: nessuna

post: RESULT è l'insieme ottenuto dall'insieme *i* aggiungendo l'elemento *e*; se *e* appartiene già a *i* allora *j* coincide con *i*

elimina(Ins i, T e) \rightarrow *Ins*

pre: nessuna

post: RESULT è l'insieme ottenuto dall'insieme *i* eliminando l'elemento *e*; se *e* non appartiene a *i* allora RESULT coincide con *i*

membro(Ins i, T e) \rightarrow Boolean

pre: nessuna

post: RESULT è true se l'elemento *e* appartiene all'insieme *i*, false altrimenti

scegli(Ins i) \rightarrow *T*

pre: l'insieme *i* non è l'insieme vuoto

post: RESULT è un elemento (qualsiasi) di *i*

FineTipoAstratto

È una astrazione ai valori (complesso)

nostro costruttore, se implementassimo in Java questa rappresentazione.

23.1.3 Astrazione di Entità:

L'altra tipologia di dato mutabile: Nota che in realtà manca il dominio delle stringhe nei do-

Tipo di dato astratto Persona

TipoAstratto Persona

Domini

Persona : dominio delle persone – dominio di interesse

Funzioni

creaPersona(String n, String r) \rightarrow *Persona*

pre: nessuna

post: RESULT è una persona che ha nome *n* e residenza *r*

nome(Persona p) \rightarrow String

pre: nessuna

post: RESULT è il nome della persona *p*

residenza(Persona p) \rightarrow String

pre: nessuna

post: RESULT è la residenza della persona *p*

cambiaResidenza(Persona p, String r) \rightarrow *Persona*

pre: nessuna

post: RESULT è la persona *p* con la residenza cambiata in *r*

FineTipoAstratto

È una astrazione di entità

mini della rappresentazione. Comunque come abbiamo detto prima, con l'esempio della Cabala, l'entità è concetto labile: se parlo di numeri parlo di valore, perché sono funzionali all'applicativo,

l'applicativo non parla di loro. Se invece l'oggetto di interesse della mia app sono i numeri, tipo app che parla della Cabala. L'entità è ciò di cui parla l'applicazione.

23.2 Schemi Realizzativi:

Vediamo come potrei realizzare questi tipi di dato astratto, cioè come passare dal dato astratto all'implementazione. Abbiamo due categorizzazioni degli schemi realizzativi:

- Con Side-Effect o Funzionali (cioè senza Side-Effect). Per **Side-Effect** si intende il caso in cui i metodi della classe operano modifiche sugli oggetti e non ne creano di nuovi modificati.
- Con o senza Condivisione di Memoria. Per **Condivisione di Memoria** in Java si intende la condivisione di memoria tra istanze della stessa classe. Quindi se due istanze parlano degli stessi valori questi non vengono replicati ma le due istanze puntano allo stesso valore. Altro esempio di condivisione di memoria: le stringhe. Ho due variabili che contengono la stessa stringa: queste due var puntano allo stesso valore nell'heap. Anche Integer, Double ecc. funzionano così (i tipi base int, double ecc. no)

In Java vale la seguente tabella:

Schema Realizzativo	Con Condivisione di Memoria	Senza Condivisione di Memoria
Con Side-Effect	NO: interferenza	SI: schema privilegiato in Java
Funzionali	SI: schema privilegiato in Java	SI: ma può sprecare memoria

Quei due sono gli schemi privilegiati da Java perché:

- con side e senza condivisione: le classi sono così. Classe Persona, facevamo side-effect per modificare la persona e le persone tra loro non condividevano informazioni.
- senza side e con condivisione: caso delle stringhe in Java.

23.2.1 Il caso dell'interferenza:

Va evitata, facciamo un esempio con le Linked List, in cui appunto usiamo side e condivisione.

```
public class ListaStringheF {

    ...
    public void setFirstInfo(String x) {
        nodoinit.info = x
    }
}
```

Questa è la classe che definisce le linked list. Istanziando ListaStringheF creiamo un nodo. Abbiamo poi il metodo nodoinit.info che ci permette di associare a questo nodo un nodo NodoLista, quindi creando appunto una linked list. Ciò significa che posso fare side-effect e modificare la mia istanza di ListaStringheF. Ma questo inoltre mi permette di creare un'altra istanza di ListaStringheF, e di usare su di lei il metodo nodoinit.info, associandogli lo stesso valore che ho associato all'istanza precedente: in questo modo ho condivisione di memoria. Vediamo come si verifica l'interferenza, se io faccio:

```
ListaStringheF l1 = new ListaStringheF().add(0, "A").add(1, "B").add(2, "C");
```

Ho creato la lista l1²⁵ che ha come nodi A B e C, in quest'ordine. Ora faccio:

```
ListaStringheF l2 = l1.add(0, "Z")
```

Quindi ho creato la lista l2 che ha come nodi il suo peculiare nodo Z e poi quelli di A, è cioè Z A B e C. E poiché c'è condivisione di memoria, A B e C non sono stati replicati ma sono gli stessi nodi che aveva l1.

```
l1.setFirstInfo("X")
```

²⁵Sulle dispense nel codice c'è la per l1 e l2 per l2

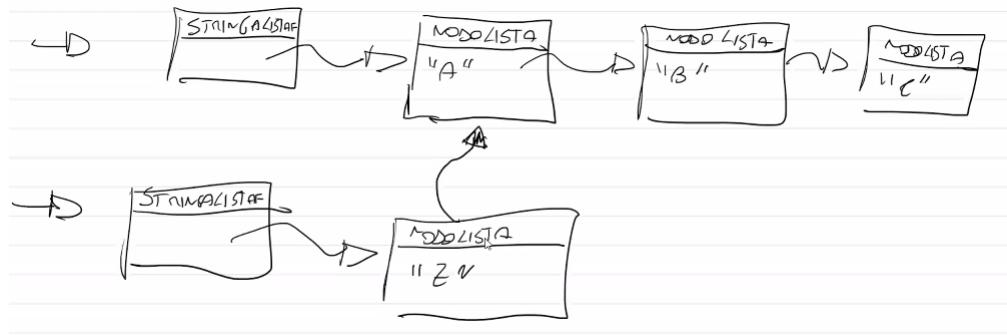
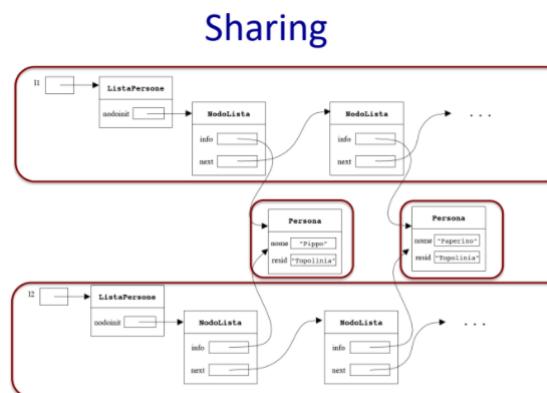


Figure 8: l1 in alto e l2 in basso.

Fa side effect sulla prima lista modificando il nodo iniziale. Che succede? Che viene modificato il nodo contenente A, *modificandolo sia per l1 che per l2!* Come risolvere questo problema? Con lo sharing:

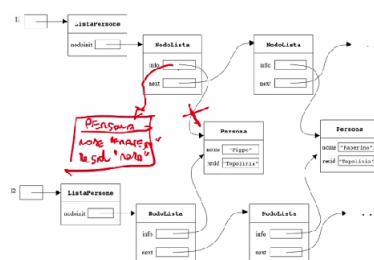
23.2.2 Sharing:

Lo sharing è il tipo di condivisione giusta da adottare. Quindi sì alla condivisione di memoria, ma solo degli oggetti, non dei nodi!! Ora vediamo meglio. Cioè quando ho una lista da un'altra



Lo sharing NON è interferenza! E' uno dei fondamenti dell' object orientation!!.

lista io replica gli elementi NodoLista e condivido solo le rappresentazioni in memoria. Notiamo una differenza con le linked list di prima: ora il campo info ha un riferimento a Persona prima ce lo aveva ad una stringa (o meglio, ad una lettera es. "A", "Z" ecc.), ma il concetto è identico e questa cosa ovviamente funziona anche con le stringhe. Comunque il discorso è che quando creo l2 non collego i nodi in comune con l1 a questi, ma li replica, quindi senza condividere la memoria. La condivisione di memoria è solo relativa agli oggetti persona. In effetti ha senso: gli oggetti che sono entità abbiamo detto che non ha senso replicarli. Questo schema implementativo, in cui condivido solo gli oggetti e non i nodi non presenta interferenza, perché se prendo un NodoLista e cambio un suo campo info, questo è visibile solo alla lista a cui appartiene al nodo, quindi o a l1 o a l2, non a tutti e due. Esempio: Il comando per fare questo sarebbe



`l1.setFirstInfo(new Persona("Francesco", "Roma"))` che dovrebbe essere proprio quello che

modifica il nodo. Vediamo quindi che l2 non vede la modifica e non c'è quindi interferenza. Vediamo che il collegamento alla vecchia Persona non viene perso perché comunque l2 lo mantiene. Bene, terminata questa parentesi su interferenza e sharing vediamo adesso se adottare e in che caso condivisione e side.

23.3 Schemi realizzativi di valore (semplice) ed entità (e valori complessi), quando usare Side e Condivisione?

Allora:

- Per lo schema realizzativo di un valore semplice si usa la condivisione di memoria (come abbiamo non ha senso replicare i valori, che ci faccio con due 4 in memoria?!) e non il side-effect (sono valori, quindi sono immutabili)
- Per lo schema relizzativo di un valore complesso (collezione) non c'è la necessità di scegliere solo uno tra i due schemi di realizzazione. Vanno bene entrambi: uno ci darà tipi mutabili e l'altro ci darà tipi immutabili. Vediamo:
 - usare la condivisione di memoria ma senza il side-effect (se voglio per forza che due liste condividano i nodi non devo permettere il side-effect, in modo da evitare l'interferenza). In questo caso creo oggetti immutabili. È come il caso del valore semplice in pratica. Effettivamente se ho due liste, l1 e l2, e quest'ultima ha i suoi nodi peculiari più i nodi che condivide con l1, se vado a modificare l1 l2 non vede il cambiamento, perché modificando il nodo di l1 in realtà ne sto creando uno nuovo, e quindi l1 cambia, mentre l2 rimane attaccato al vecchio nodo, come unico collegamento a lui ora. Quindi non è che abbiamo modificato l1: abbiamo proprio creato una nuova lista l1 (proprio come gli oggetti immutabili).
 - senza condivisione di memoria (così posso usare il side-effect per fare oggetti immutabili ed evitare interferenza) e con side-effect. Cioè è il caso dello SHARING, in cui i nodi non sono condivisi e il side effect mi permette di modificare i campi info dei singoli nodi, che nel nostro caso erano campi info che puntavano a Persone. Attenzione che il side effect è sui campi info, non sulle persone: come nell'esempio dello sharing, quando faccio side effect *creo un'altra persona*, facendo side-effect sul campo info. In questo caso creo oggetti mutabili. È come il caso dell'entità in pratica.
- Per lo schema realizzativo di un entità si fa senza condivisione di memoria (anche se due Persone hanno gli stessi campi, magari stesso nome cognome e anno di nascita, non è detto che siano la stessa persona [diversamente dal caso dei numeri che sono valori], perché potrebbero essere omonimi nati lo stesso giorno, e quindi non possiamo far puntare due riferimenti alla stessa persona ma dobbiamo creare due Persone) e con side effect, perché sono oggetti mutabili.

23.4 Equals() e Clone() in questi quattro schemi realizzativi:

Astrazione di valore semplice: ricordiamo che usa uno schema di realizzazione con condivisione di memoria e funzionale. Quindi:

- equals() va ridefinito, per l'uguaglianza profonda, che nel caso dei valori, semplici o complessi che siano, va fatta. hashCode() va ridefinito perché equals() viene ridefinito.
- clone() non va ridefinito, perché non ci servono copie per i valori (non ci faccio nulla con due 4 in memoria) ma condivisione di memoria.

Astrazione di Entità: ricordiamo che usa uno schema di realizzazione senza condivisione di memoria e con side-effect. Quindi:

- equals() non va ridefinito: per vedere se due entità sono uguali ci serve l'uguaglianza superficiale, poiché due Persone sono uguali se e solo se sono la stessa persona, non se hanno gli stessi campi (potrebbero essere omonimi che sono nati lo stesso giorno). hashCode() non va ridefinito perché equals() non viene ridefinito.
- clone() non va ridefinito, perché non ha senso copiare delle entità (che ci faccio con il clone di una Persona?), anche perché in questo modo non avremmo clonato la Persona, ma per quanto detto subito sopra ne avremmo creata una nuova.

Astrazione di Valore Collezione: ricordiamo che qui abbiamo due schemi di realizzazione possibili:

- Con condivisione di memoria e senza side-effect:
 - equals() va ridefinito, poiché per le liste in ogni caso dobbiamo verificare che abbiamo gli stessi elementi, quindi dobbiamo testare l'uguaglianza profonda. hashCode() quindi anche va ridefinito
 - clone() non va ridefinito se c'è la condivisione di memoria, proprio perché quello di default fa un assegnazione di riferimento allo stesso oggetto, cioè implica condivisione di memoria, che a noi va bene.
- Senza condivisione di memoria e con side-effect:
 - equals() va ridefinito, poiché per le liste in ogni caso dobbiamo verificare che abbiamo gli stessi elementi, quindi dobbiamo testare l'uguaglianza profonda. hashCode() quindi anche va ridefinito. Questo vale in tutti e due gli schemi quindi.
 - clone() va ridefinito se non c'è la condivisione di memoria, perché quello di default fa un assegnazione di riferimento allo stesso oggetto, quindi ci ritroviamo due riferimenti che CONDIVIDONO MEMORIA, e a noi non va bene.

Per ulteriori approfondimenti leggi le 28 pagine del libro su sta roba che trovi nella cartella drive del corso, precisamente nella sezione JavaParte2. Si tratta del capitolo 13 del libro di Java.

https://drive.google.com/drive/folders/1Ua6A7HiZOK9Ww1oPBf97ch-Xo_SKxdJ3

23.5 Implementazioni FC, SS, SC di Insieme:

23.5.1 Implementazione FC:

Vogliamo descrivere l'implementazione Funzionale (senza side-effect) e Con Condivisione di Memoria della collezione Insieme.

```
public class InsiemeFC
```

Vediamo innanzitutto l'utilizzo di una classe non public, utilizzata dal programmatore come ausilio per la creazione della classe public del file: è proprio così che la utilizziamo. Si tratta di una struttura dati, eccola:

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

In questo modo posso ho una linked list promiscua: Object è generico, un nodo potrà avere una info Persona, un altro edificio ecc.

```
class NodoLista<T> {  
    T info;  
    NodoLista<T> next;  
}
```

In questo modo vincolo il tipo ad essere uno solo per tutta la lista. Bene andiamo a implementare il tipo di dato astratto ora. Ricordiamocelo: Solo che, come abbiamo detto, gli ADT sono sempre scritti in maniera funzionale, mentre noi adesso andremo a implementare Insieme in un linguaggio OOP, cioè Java. Questo significa che mentre l'ADT ha metodi che prendono come input l'insieme su cui operare, per noi questo insieme sarà l'oggetto di invocazione. Ok, andiamo a implementarlo con F e C:

```
public class InsiemeFC {
```

Tipo di dato astratto

Insieme(T)

TipoAstratto *Insieme(T)*

Domini

Ins : dominio di interesse

T : dominio degli elementi dell'insieme

Funzioni

insiemeVuoto() → Ins

pre: nessuna

post: RESULT è l'insieme vuoto

estVuoto(Ins i) → Boolean

pre: nessuna

post: RESULT è true se i è il valore corrispondente all'insieme vuoto, false altrimenti

inserisci(Ins i, T e) → Ins

pre: nessuna

post: RESULT è l'insieme ottenuto dall'insieme i aggiungendo l'elemento e ; se e

appartiene già a i allora j coincide con i

elimina(Ins i, T e) → Ins

pre: nessuna

post: RESULT è l'insieme ottenuto dall'insieme i eliminando l'elemento e ; se e non appartiene a i allora RESULT coincide con i

membro(Ins i, T e) → Boolean

pre: nessuna

post: RESULT è true se l'elemento e appartiene all'insieme i , false altrimenti

scegli(Ins i) → T

pre: l'insieme i non è l'insieme vuoto

post: RESULT è un elemento (qualsiasi) di i

FineTipoAstratto

E una astrazione ai valori (compleso)

```

private NodoLista inizio; //ci teniamo il riferimento alla lista

public InsiemeFC() { //costruttore che fa la stessa cosa di quello di default:
    potevamo non metterlo ma lo facciamo per leggibilità'. Possiamo vedere che
    insiemeVuoto() nell'ADT, corrisponde a questo costruttore
    inizio = null;
}

public boolean estVuoto() { //corrisponde a estVuoto() nell'ADT
    return inizio == null;
}

public InsiemeFC inserisci(Object e) { //corrisponde a inserisci() nell'ADT
    if (appartiene(e, inizio)) {
        return this; //se l'oggetto e si trova già' nella lista ti restituisco
                     l'insieme su cui hai invocato il metodo inserisci, cioe' ti invio
                     l'insieme su cui operavi per inserire l'elemento.
    } else { //se invece l'elemento non si trova nell'insieme. Con una lista collegata
             il modo di inserirlo e' metterlo all'inizio.

        InsiemeFC newSet = newInsiemeFC();
        NodoLista newNode = new NodoLista();
        newNode.info = e;
        newNode.next = inizio;
        newSet.inizio = newNode //newNode e' il nostro nuovo primo nodo
        return newSet;
    }
}

private static boolean appartiene(Object e, NodoLista l) { //metodo ausiliario per
}

```

```

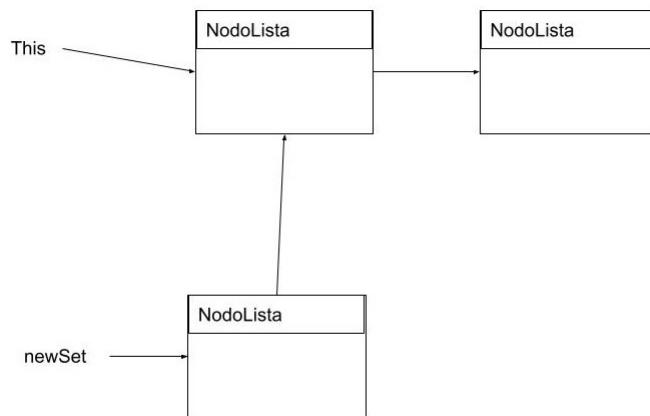
inserisci()

    return (l != null) && (l.info.equals(e) || appartiene(e, l.next)); //ricorsivo:
        l'elemento e sta nel nodo l della lista? Se no vai avanti nella lista a
        cercarlo

}

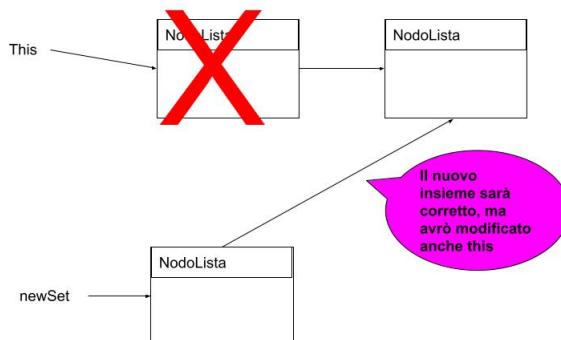
```

Quindi, graficamente, se abbiamo un nuovo elemento da inserire nell'insieme, questo è ciò che accade quando usiamo inserisci: Notare la condivisione di memoria, dato che newSet si connette



tramite il suo unico NodoLista ai NodoLista di this, this che è l'istanza di InsiemeFC su cui abbiamo chiamato il metodo inserisci(). Bene, ora occupiamoci dei metodi rimasti:

- metodo elimina() nell'ADT. Attenzione perché con condivisione di memoria e senza side-effect, il metodo mi deve restituire una nuova linked list newSet dove non c'è più il nodo, senza che questo si ripercuoti sulla linked list this, originaria. Quindi il semplice bypass mi restituisce un newSet corretto ma mi modifica anche this, l'insieme su cui invoco il metodo. Come fare allora? Per fare questo usiamo dei metodi statici ausiliari.



```

private static NodoLista cancellaFC(Object e, NodoLista l) { //cancella da una
    lista collegata un nodo lista e restituisce come output il riferimento alla
    nuova lista senza l'oggetto. Si tratta di un METODO AUSILIARIO per elimina()

    if (l == null) {

```

```

        return null
    } else if (l.info.equals(e)) {

        return l.next;

    } else {
        NodoLista ll = new NodoLista();
        ll.info = l.info;
        ll.next = cancellaFC(e, l.next) //ricorsività
        return ll;
    }

}

```

E così abbiamo risolto il problema dell'interferenza. Come? Abbiamo fatto condivisione solo degli elementi successivi a quello da eliminare, mentre quelli prima li abbiamo clonati, e l'elemento stesso non lo abbiamo aggiunto.

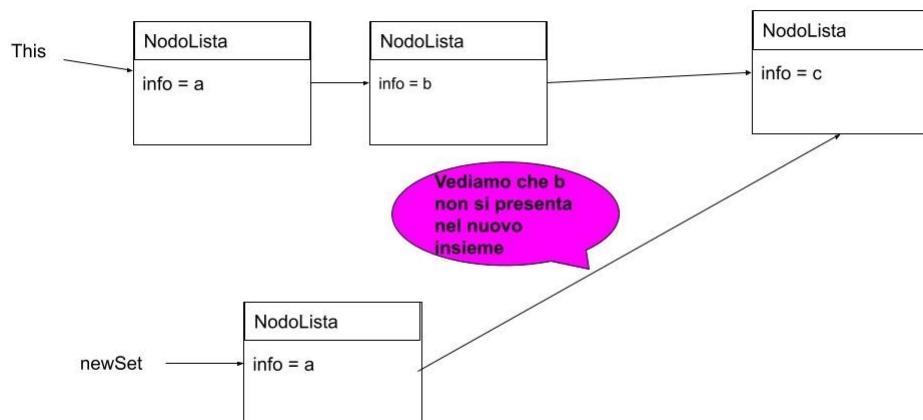


Figure 9: Nel caso in cui si voglia eliminare il nodo con l'informazione b

Notiamo che, nel caso in cui l'elemento che si vuole eliminare non sia nella lista, la lista viene semplicemente clonata. Possiamo, invece di sprecare memoria in una clonazione simile, fare una verifica preventiva per vedere se l'elemento è presente nella lista, e se non lo è, restituire la stessa lista su cui è stato invocato il metodo `elimina()`, con `this`. Perciò il nostro metodo `elimina()` può essere scritto così:

```

public InsiemeFC elimina(Object e) {

    if (!appartiene(e, inizio)) {
        return this;
    } else {
        InsiemeFC newSet = new InsiemeFC();
        newSet.inizio = cancellaFC(e, inizio);
        return newSet;
    }
}

```

- metodo membro() nell'ADT:

```

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

```

- metodo scegli() nell'ADT:

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

Bene, abbiamo tutti i metodi dell'insieme implementato Funzionalmente e Con Condivisione di Memoria. Ok, quindi il nostro InsiemeFc è:

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

public class InsiemeFC {

    private NodoLista inizio; //ci teniamo il riferimento alla lista

    public InsiemeFC() { //costruttore che fa la stessa cosa di quello di default:
        potevamo non metterlo ma lo facciamo per leggibilità'. Possiamo vedere che
        insiemeVuoto() nell'ADT, corrisponde a questo costruttore
        inizio = null;
    }

    public boolean estVuoto() { //corrisponde a estVuoto() nell'ADT
        return inizio == null;
    }

    public InsiemeFC inserisci(Object e) { //corrisponde a inserisci() nell'ADT
        if (appartiene(e, inizio)) {
            return this; //se l'oggetto e si trova già' nella lista ti restituisco
            l'insieme su cui hai invocato il metodo inserisci, cioè ti invio
            l'insieme su cui operavi per inserire l'elemento.
        } else { //se invece l'elemento non si trova nell'insieme. Con una lista collegata
            il modo di inserirlo e' metterlo all'inizio.

            InsiemeFC newSet = newInsiemeFC();
            NodoLista newNode = new NodoLista();
            newNode.info = e;
            newNode.next = inizio;
            newSet.inizio = newNode //newNode e' il nostro nuovo primo nodo
            return newSet;
        }
    }

    private static boolean appartiene(Object e, NodoLista l) { //metodo ausiliario per
        inserisci()

        return (l != null) && (l.info.equals(e) || appartiene(e, l.next)); //ricorsivo:
        l'elemento e sta nel nodo l della lista? Se no vai avanti nella lista a
        cercarlo
    }

    private static NodoLista cancellaFC(Object e, NodoLista l) { //cancella da una lista
        collegata un nodo lista e restituisce come output il riferimento alla nuova lista
    }
}
```

```

    senza l'oggetto. Si tratta di un METODO AUSILIARIO per elimina()

    if (l == null) {
        return null
    } else if (l.info.equals(e)) {

        return l.next;

    } else {
        NodoLista l1 = new NodoLista();
        l1.info = l.info;
        l1.next = cancellaFC(e, l.next) //ricorsività
        return l1;
    }
}

public InsiemeFC elimina(Object e) {

    if (!appartiene(e, inizio)) {
        return this;
    } else {
        InsiemeFC newSet = new InsiemeFC();
        newSet.inizio = cancellaFC(e, inizio);
        return newSet;
    }
}

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

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

```

Ed eccoci qua. Vediamo che, se non fosse che l'ADT prende come parametri gli insiemi che poi modifica poiché ha un approccio funzionale (nel senso di Python, non nel senso di Side-Effect), mentre la nostra implementazione in Java è OOP, e quindi gli insiemi che modifica non sono passati come parametri ma sono gli oggetti di invocazione dei metodi, l'ADT e questa implementazione si assomigliano molto. Comunque, una cosa che possiamo fare per migliorare il nostro codice è sostituire ai metodi statici della classe InsiemeFC dei metodi non statici di NodoLista.

```

class NodoLista {
    Object info;
    NodoLista next;

    boolean appartiene(Object e) { //abbiamo trasformato il metodo statico di InsiemeFC
        in un metodo di istanza di NodoLista

        return (this.info.equals(e) || this.next.appartiene(e));
    }

    NodoLista cancellaFC(Object e) {

        if (this.info.equals(e)) {

```

```

        return this.next;

    } else {
        NodoLista l1 = new NodoLista();
        l1.info = this.info;
        l1.next = this.next.cancellaFC(e);
        return l1;
    }
}

}

public class InsiemeFC {

    private NodoLista inizio;

    public InsiemeFC() {
        inizio = null;
    }

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

    public InsiemeFC inserisci(Object e) {
        if (inizio == null || inizio.appartiene(e)) {
            return this;
        } else {

            InsiemeFC newSet = newInsiemeFC();
            NodoLista newNode = new NodoLista();
            newNode.info = e;
            newNode.next = inizio;
            newSet.inizio = newNode

        }
    }

    public InsiemeFC elimina(Object e) {

        if (inizio == null || !inizio.appartiene(e)) {
            return this;
        } else {
            InsiemeFC newSet = new InsiemeFC();
            newSet.inizio = inizio.cancellaFC(e);
            return newSet;
        }
    }

    public boolean membro(Object e) {
        return inizio != null && inizio.appartiene(e);
    }

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

```

```
    }  
}
```

Vediamo che abbiamo rimosso i controlli su null, perché this è sempre non null. Inoltre abbiamo dovuto sostituire alle chiamate funzionali dei metodi statici delle invocazioni di metodi su istanze di NodoLista. Questa versione è completamente OO, ergo più pulita: quando si può è meglio evitare i metodi statici. Bene, abbiamo migliorato il nostro codice eliminando i metodi statici, adesso facciamo un altro passo e aggiungiamo i generics.

```
class NodoLista<T> {  
    T info;  
    NodoLista<T> next;  
  
    boolean appartiene(T e) { //abbiamo trasformato il metodo statico di InsiemeFC in un  
        //metodo di istanza di NodoLista  
  
        return (this.info.equals(e) || (this.next != null && this.next.appartiene(e)));  
        //questo equals e' l>equals della classe T, quindi se sara' un'uguaglianza  
        //superficiale o profonda dipendera' dalla classe T  
    }  
  
    NodoLista<T> cancellaFC(T e) {  
  
        if (this.info.equals(e)) {  
  
            return this.next;  
  
        } else {  
            NodoLista<T> l1 = new NodoLista<T>();  
            l1.info = this.info;  
            if (this.next != null) {  
                l1.next = this.next.cancellaFC(e);  
            }  
            return l1;  
        }  
    }  
}  
  
public class InsiemeFC<T> {  
  
    private NodoLista<T> inizio;  
  
    public InsiemeFC() {  
        inizio = null;  
    }  
  
    public boolean estVuoto() {  
        return inizio == null;  
    }  
  
    public InsiemeFC<T> inserisci(T e) {  
        if (inizio != null && inizio.appartiene(e)) {  
            return this;  
        } else {  
  
            InsiemeFC<T> newSet = new InsiemeFC<T>();  
            NodoLista<T> newNode = new NodoLista<T>();  
            newNode.info = e;
```

```

        newNode.next = inizio;
        newSet.inizio = newNode;
        return newSet;
    }

}

public InsiemeFC<T> elimina(T e) {

    if (inizio == null || !inizio.appartiene(e)) {
        return this;
    } else {
        InsiemeFC<T> newSet = new InsiemeFC<T>();
        newSet.inizio = inizio.cancellaFC(e);
        return newSet;
    }
}

public boolean membro(T e) {
    return inizio != null && inizio.appartiene(e);
}

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

//andiamo a implementare equals() e hashCode(), clone() non va overridato perche'
//c'e' Condivisione di Memoria

@Override
public boolean equals(Object o) {
    if (o == null || !(getClass().equals(o.getClass()))) {
        return false;
    } else {
        InsiemeFC<T> ins = (InsiemeFC<T>) o;
        NodoLista<T> l = inizio;

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

        while (l != null) { //non basta solo vedere se ins ha tutti gli elementi di l,
                           //bisogna vedere anche il contrario: le due liste devono essere UGUALI

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

```

```

@Override
public int hashCode() { //il modo migliore di farlo sarebbe ridefinire hashCode
    all'interno di nodolista e poi qui fare una combinazione lineare di hashCode

    return 1; //qui famo sta cosa per sbrigarsi. Si tratta di una cosa corretta
              sintatticamente, perche' se restituisco sempre 1 e' ovvio che i due valori
              avranno sempre stesso hashCode quando confrontati

}

@Override
public String toString() {
    String s = "(" ;
    NodoLista l = inizio;
    while (l != null) {
        s = s + l.info + " ";
        l = l.next;
    }
    s = s + ")";
    return s;
}

```

```

public class Main
{
    public static void main(String[] args) {

        InsiemeFC<String> ins1 = new InsiemeFC<String>();

        System.out.print("l'insieme ins1 vuoto? ");
        System.out.println(ins1.estVuoto());


        InsiemeFC<String> ins2 = ins1.inserisci("Ciao");
        ins2 = ins2.inserisci("Hello");

        System.out.print("l'insieme ins1 vuoto? ");
        System.out.println(ins1.estVuoto());

        System.out.print("l'insieme ins2 vuoto? ");
        System.out.println(ins2.estVuoto());


        InsiemeFC<String> ins3 = ins2.elimina("Hello");

        System.out.print("l'insieme ins3 vuoto? ");
        System.out.println(ins3.estVuoto());


        System.out.println("\n#####\n");

        System.out.print("l'insieme ins1 contiene 'Hello'? ");
        System.out.println(ins1.membro("Hello"));

        System.out.print("l'insieme ins2 contiene 'Hello'? ");
        System.out.println(ins2.membro("Hello"));

        System.out.print("l'insieme ins3 contiene 'Hello'? ");
        System.out.println(ins3.membro("Hello"));

        System.out.println("\n#####\n");
    }
}

```

```

        System.out.print("Prendiamo il primo elemento di ins1... ");
        System.out.println(ins1.scegli());

        System.out.print("Prendiamo il primo elemento di ins2... ");
        System.out.println(ins2.scegli());

        System.out.print("Prendiamo il primo elemento di ins3... ");
        System.out.println(ins3.scegli());

        System.out.println("\n#####\n");

        System.out.println("Testiamo ora equals()");

        InsiemeFC<Number> ins4 = new InsiemeFC<Number>();

        ins4 = ins4.inserisci(21);
        ins4 = ins4.inserisci(1);
        ins4 = ins4.inserisci(2);
        ins4 = ins4.inserisci(212);
        ins4 = ins4.inserisci(2);

        InsiemeFC<Number> ins5 = new InsiemeFC<Number>();

        ins5 = ins5.inserisci(21);
        ins5 = ins5.inserisci(1);
        ins5 = ins5.inserisci(2);
        ins5 = ins5.inserisci(212);
        ins5 = ins5.inserisci(2);

        InsiemeFC<Number> ins6 = new InsiemeFC<Number>();

        ins6 = ins6.inserisci(21);
        ins6 = ins6.inserisci(1);
        ins6 = ins6.inserisci(2);
        ins6 = ins6.inserisci(212);
        ins6 = ins6.inserisci(2);
        ins6 = ins6.inserisci(0);

        System.out.print("ins4 uguale profondamente a ins5? ");
        System.out.println(ins4.equals(ins5));

        System.out.print("ins5 uguale profondamente a ins6? ");
        System.out.println(ins5.equals(ins6));

        System.out.print("ins6 uguale profondamente a ins6? ");
        System.out.println(ins6.equals(ins6));

        System.out.print("\n#####\n");

        System.out.println(ins1);
        System.out.println(ins2);
        System.out.println(ins3);
        System.out.println(ins4);
        System.out.println(ins5);
        System.out.println(ins6);

    }
}

```

In questo modo abbiamo trasformato un insieme di oggetti potenzialmente promiscui in un insieme di oggetti dello stesso tipo.

23.5.2 Implementazione SS:

Quindi con Side-Effect e senza Condivisione di Memoria. Lo facciamo direttamente con i generics e senza metodi statici, ma sulla cartella Google Drive del corso trovi anche la versione senza Generics (che quindi usa Object) e con i metodi statici.

```
class NodoLista<T> {
    T info;
    NodoLista<T> next;

    boolean appartiene(T e) { //sia con sia senza condivisione non cambia nulla
        return (this.info.equals(e) || (this.next != null && this.next.appartiene(e)));
    }

    NodoLista<T> cancellaSS(T e) { //cambia: non essendoci condivisione devo modificare
        la lista quando devo modificare l'insieme

        if (this.info.equals(e)) {

            return this.next;

        } else {
            this.next = this.next.cancellaSS(e);
            return this;
        }
    }

    NodoLista<T> copiaLista() { //funzione ausiliaria di clone()
        NodoLista<T> inizioNuovaLista = new NodoLista<T>();
        inizioNuovaLista.info = this.info;
        if (this.next != null) {
            inizioNuovaLista.next = this.next.copiaLista();
        }
        return inizioNuovaLista;
    }
}

public class InsiemeSS<T> implements Cloneable {

    private NodoLista<T> inizio;

    public InsiemeSS() {
        inizio = null;
    }

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

    public void inserisci(T e) { //restituisce void perche' c'e' side-effect
        if (inizio == null || !inizio.appartiene(e)) {

            NodoLista<T> newNode = new NodoLista<T>();
            newNode.info = e;
            newNode.next = inizio;
            inizio = newNode;
        }
    }
}
```

```

    }

    public void elimina(T e) {

        if (inizio != null && !inizio.appartiene(e)) {
            inizio = inizio.cancellaSS(e);
        }
    }

    public boolean membro(T e) {
        return inizio != null && inizio.appartiene(e);
    }

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

    public boolean equals(Object o) { //equals rimane uguale
        if (o == null || !(getClass().equals(o.getClass()))) {
            return false;
        } else {
            InsiemeSS<T> ins = (InsiemeSS<T>) o;
            NodoLista<T> l = inizio;

            while (l != null) {
                if (ins.inizio == null || !ins.inizio.appartiene(l.info)) {
                    return false;
                }
                l = l.next;
            }
            l = ins.inizio;

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

    @Override
    public int hashCode() {
        return 1;
    }

    @Override
    public Object clone() { //devo ridefinire clone, per evitare condivisione di memoria

        try {
            InsiemeSS ins = (InsiemeSS) super.clone(); //per fare la copia superficiale
            delle variabili

            if (inizio != null) {
                ins.inizio = this.inizio.copiaLista();
            }
            return ins;
        } catch (CloneNotSupportedException e) {

```

```

        throw new InternalError(e.toString());
    }
}

@Override
public String toString() { // overriding di toString()
    String s = "(";
    NodoLista l = inizio;
    while (l != null) {
        s = s + l.info + " ";
        l = l.next;
    }
    s = s + ")";
    return s;
}
}

-----

```

```

public class Main
{
    public static void main(String[] args) {

        InsiemeSS<String> ins1 = new InsiemeSS<String>();

        System.out.print("l'insieme ins1 vuoto? ");
        System.out.println(ins1.estVuoto());


        InsiemeSS<String> ins2 = new InsiemeSS<String>();
        ins2.inserisci("Ciao");
        ins2.inserisci("Hello");

        System.out.print("l'insieme ins1 vuoto? ");
        System.out.println(ins1.estVuoto());

        System.out.print("l'insieme ins2 vuoto? ");
        System.out.println(ins2.estVuoto());


        InsiemeSS<String> ins3 = (InsiemeSS<String>) ins2.clone();

        System.out.print("l'insieme ins3 vuoto? ");
        System.out.println(ins3.estVuoto());


        System.out.println("\n#####\n");

        System.out.print("l'insieme ins1 contiene 'Hello'? ");
        System.out.println(ins1.membro("Hello"));

        System.out.print("l'insieme ins2 contiene 'Hello'? ");
        System.out.println(ins2.membro("Hello"));

        System.out.print("l'insieme ins3 contiene 'Hello'? ");
        System.out.println(ins3.membro("Hello"));

        System.out.println("\n#####\n");

        System.out.print("Prendiamo il primo elemento di ins1... ");
        System.out.println(ins1.scegli());
    }
}

```

```

        System.out.print("Prendiamo il primo elemento di ins2... ");
        System.out.println(ins2.scegli());

        System.out.print("Prendiamo il primo elemento di ins3... ");
        System.out.println(ins3.scegli());

        System.out.println("\n#####\n");

        System.out.println("Testiamo ora equals()");

        InsiemeSS<Number> ins4 = new InsiemeSS<Number>();

        ins4.inserisci(21);
        ins4.inserisci(1);
        ins4.inserisci(2);
        ins4.inserisci(212);
        ins4.inserisci(2);

        InsiemeSS<Number> ins5 = (InsiemeSS<Number>) ins4.clone();

        InsiemeSS<Number> ins6 = new InsiemeSS<Number>();

        ins6.inserisci(21);
        ins6.inserisci(1);
        ins6.inserisci(2);
        ins6.inserisci(212);
        ins6.inserisci(2);
        ins6.inserisci(0);

        System.out.print("ins4 uguale profondamente a ins5? ");
        System.out.println(ins4.equals(ins5));

        System.out.print("ins4 uguale superficialmente a ins5? ");
        System.out.println(ins4 == ins5);

        System.out.print("ins5 uguale profondamente a ins6? ");
        System.out.println(ins5.equals(ins6));

        System.out.print("ins5 uguale superficialmente a ins6? ");
        System.out.println(ins5 == ins6);

        System.out.print("ins6 uguale profondamente a ins6? ");
        System.out.println(ins6.equals(ins6));

        System.out.print("ins6 uguale superficialmente a ins6? ");
        System.out.println(ins6 == ins6);

        System.out.print("\n#####\n");

        System.out.println(ins1);
        System.out.println(ins2);
        System.out.println(ins3);
        System.out.println(ins4);
        System.out.println(ins5);
        System.out.println(ins6);

    }
}

```

Quindi quali sono le differenze tra FC e SS? In FC non ho bisogno di ridefinire il clone perché qualsiasi operazione di modifica che faccio sul mio oggetto questa si comporterà correttamente, perché condividerà la memoria che può e scarterà il resto. Invece con SS, se devo creare delle copie devo per forza ridefinire clone().

23.5.3 Implementazione SC???

23.6 Implementazione di tipo di dato astratto studente:

Si tratta di un'entità, quindi si fa sempre con side effect e senza condivisione di memoria. equals() e clone() non vanno quindi ridefiniti. Vediamo che i domini nell'ADT corrispondono a classi nell'implementazione in Java! Tiriamo innanzitutto fuori il diagramma delle classi

```

TipoAstratto Studente
Domini
    Studente : dominio degli studenti – dominio di interesse
    Esame : dominio degli esami
    PianoDiStudio : dominio dei piani di studio
Funzioni
    creaStudente(Stringa n, Stringa f, PianoDiStudi p) ↦ Studente
        pre: nessuna
        post: RESULT è uno studente avente come nome n, iscritto alla facoltà f, avente
              come piano di studi p e che non ha superato alcun esame
    nome(Studente s) ↦ Stringa
        pre: nessuna
        post: RESULT è il nome dello studente s
    facolta(Studente s) ↦ Stringa
        pre: nessuna
        post: RESULT è il la stringa che rappresenta la facoltà a cui lo studente s è iscritto
    piano(Studente s) ↦ PianoDiStudi
        pre: nessuna
        post: RESULT è il piano di studi dello studente s
    assegnaFacoltà(Studente s, Stringa f) ↦ Studente
        pre: nessuna
        post: RESULT è lo studente s con la facoltà a cui è iscritto modificata in f
    media(Studente s) ↦ Reale
        pre: nessuna
        post: RESULT è la media dei voti ottenuti dallo studente s negli esami sostenuti
    votoEsame(Studente s, Esame e) ↦ Intero
        pre: nessuna
        post: RESULT è il voto ottenuto dallo studente s nell'esame e, se effettivamente
              superato; altimenti RESULT è pari a 0
FineTipoAstratto

```

in UML: Questo ci ha chiarito molto le idee, adesso sappiamo che classi creare e come costruire l'implementazione SS in Java:

```

package studente;

import esame.*;
import pianodistudi.*;

//classe ausiliaria per rappresentare gli esami con voto

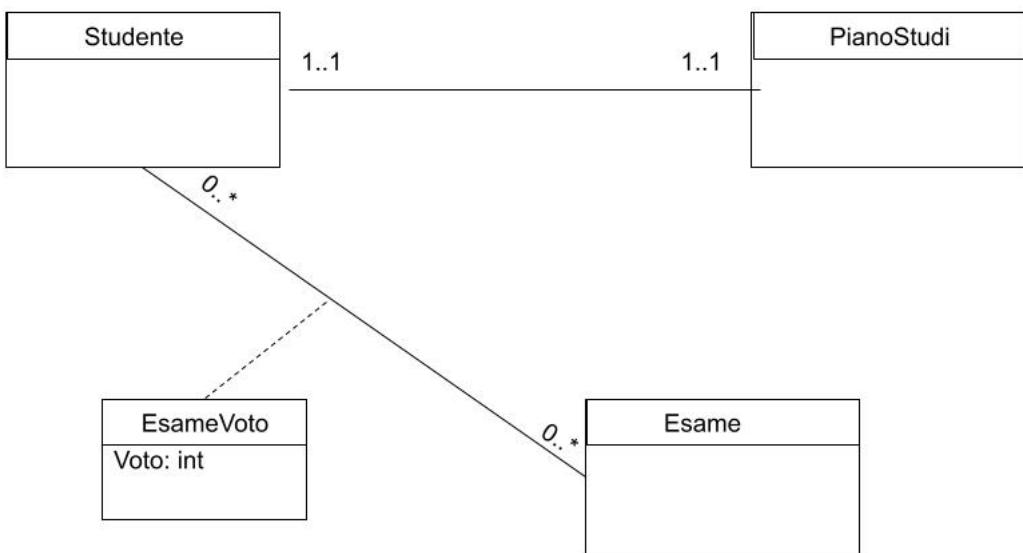
class EsameVoto { //EsameVoto e' la classe che rappresenta, nel nostro diagramma UML, la
    classe associazione tra Studente ed Esame. Dobbiamo fare cosi' perche' non ha senso
    mettere direttamente il campo dati voto nella classe Esame (cioe' nel diagramma UML
    mettere l'attributo voto nella classe Esame), non avendo l'esame per definizione la
    possibilita' di prendere solo un numero come voto (ma bensi' ogni numero da 0 a 31);
    questo significa che il voto e' una proprieta' dell'esame associato a un particolare
    studente, e quindi e' una classe associazione

    private Esame es;
    private int vo;

    public EsameVoto(Esame e, int v) {
        es = e;
        vo = v;
    }

    public Esame esame() {

```



```

        return es;
    }

    public int voto() {
        return vo;
    }
}

public class Studente {

    // rappresentazione dell'oggetto

    private String nome;
    private String facolta;
    private PianoDiStudi piano_studi;
    private EsameVoto[] esamiFatti; // dobbiamo rappresentare l'insieme degli esami
                                    // per semplicita' usiamo un semplice array
                                    // ma potremmo usare, e.g., un InsiemeSS
    private int numEsami;

    // realizzazione delle funzioni del tipo astratto

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

```

public String nome() {
    return nome;
}

public String facolta() {
    return facolta;
}

public PianoDiStudi piano() {
    return piano_studi;
}

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

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

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

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

// equals(Object) e hashCode () ereditate da Object sono corrette
// anche clone() va bene inaccessibile.
}

```

```

package esame;

public class Esame {
    private String nomeEsame;

    public Esame(String id) {
        nomeEsame = id;
    }

    public String toString() {
        return nomeEsame;
    }
}

```

```

package pianodistudi;

public class PianoDiStudi {
    private String id;

```

```

public PianoDiStudi() {
    id = "Piano di studi";
}

public String toString() {
    return id + " " + super.toString();
}
}

-----
package provastudente;

import esame.*;
import pianodistudi.*;
import studente.*;

public class ProvaStudente {
    public static void main(String[] args) {
        PianoDiStudi p1 = new PianoDiStudi();
        System.out.println(p1);
        Esame fond1 = new Esame("Fondamenti 1");
        System.out.println(fond1);
        Esame fond2 = new Esame("Fondamenti 2");
        System.out.println(fond2);
        Esame fisical1 = new Esame("Fisica 1");
        System.out.println(fisical1);

        Studente paolo = new Studente("Paolo", "ingegneria", p1);
        System.out.println(paolo);
        System.out.println("Piano di studi di " + paolo + ": " + paolo.piano());
        paolo.inserisciEsame(fond1, 20);
        System.out.println(fond1 + ": " + paolo.votoEsame(fond1));
        System.out.println("Media: " + paolo.media());
        paolo.inserisciEsame(fond2, 28);
        System.out.println(fond2 + ": " + paolo.votoEsame(fond2));
        System.out.println("Media: " + paolo.media());
    }
}

```

Quindi in Java ci troviamo con 4 classi (Studente, Esame, PianoStudi e la classe associazione EsameVoto) e un main. Notare però che questa implementazione da quanto dice il professore fa pietà, perché non abbiamo ancora gli strumenti e lo schema adatto per fare la giusta implementazione. Quindi all'esame non facciamo una roba così, ma molto meglio.

23.6.1 Postilla: come avevo fatto io inizialmente questo codice:

Quello che c'è sopra è il codice presente sul Drive del corso, con grafico UML annesso. Io però prima avevo fatto l'esercizio col codice che segue, probabilmente sbagliando soprattutto per aver messo voto come attributo di esame (avendo messo voto come campo dati della classe Esame in Java), però bho non mi andava di buttarlo quindi lo lascio qua, magari servirà:

```

public class Studente {

    //campi dati
    private String nome;
    private String facolta;
    private PianoDiStudi pianoStudi;
    private Esame[] esamiFatti;
    private int numEsamiFatti; //necessario per poter inserire nuovi esami nell'array

```

```

//costruttori
public Studente(String nome, String facolta, PianoDiStudi pianoStudi) {

    this.nome = nome;
    this.facolta = facolta;
    this.pianoStudi = pianoStudi;
    this.esamiFatti = new Esame[30];
    this.numEsamiFatti = 0;
}

//campi funzione

//metodi getter

public String getNome() {
    return this.nome;
}

public String getFacolta() {
    return this.facolta;
}

public PianoDiStudi getPiano() {
    return this.pianoStudi;
}

public int getNumEsamiFatti() {

    return this.numEsamiFatti;
}

//metodi setter

public void setFacolta(String facolta) {
    this.facolta = facolta;
}

public void setEsame(String nomeEsame, int votoEsame) { //questo non c'era nel tipo
    astratto, ma io credo sia fondamentale

    this.esamiFatti[numEsamiFatti] = new Esame(nomeEsame, votoEsame);

    numEsamiFatti++;
}

//altri metodi

public double media() {

    double sum = 0;

    for (int i = 0; i < numEsamiFatti; i++) {

        sum += esamiFatti[i].getVotoEsame();
    }

    return sum / numEsamiFatti;
}

```

```

public int VotoEsame(String nomeEsame) { //qua ho messo che tu mi dai il nome io ti
    do il voto che ha preso, non ho messo quindi come parametro un tipo Esame

    for (int i = 0; i < numEsamiFatti; i++) {

        if (this.esamiFatti[i].getNomeEsame().equals(nomeEsame)) { //ma qua non c'e'
            condivisione di memoria??
                return this.esamiFatti[i].getVotoEsame();
        }
    }

    System.out.println("Lo studente non ha ancora superato l'esame");

    return 0;
}

//equals() e clone() non vanno ridefiniti nelle entita'
}

-----


public class Esame {

    //campi dati
    private String nomeEsame;
    private int votoEsame;

    //costruttori
    public Esame(String nomeEsame, int votoEsame) { //questo non c'era nel tipo astratto,
        ma io credo sia fondamentale

        this.nomeEsame = nomeEsame;
        this.votoEsame = votoEsame;
    }

    //campi funzione

    //metodi getter

    public int getVotoEsame() {
        return votoEsame;
    }

    public String getNomeEsame() {
        return nomeEsame;
    }

}

```

```

public class PianoDiStudi {

    //campi dato

    private String[] esamiDelPiano;

    //costruttore
    public PianoDiStudi(String[] esamiDelPiano) {

        this.esamiDelPiano = esamiDelPiano;
    }

    ///campi funzione

    @Override
    public String toString() {

        String s = "(";

        for (int i = 0; i < 30; i++) {

            s = s + esamiDelPiano[i] + " ";
        }

        s = s + ")";

        return s;
    }

}

-----
public class Main {

    public static void main(String[] args) {

        String[] esamiTotali = new String[30];
        for (int i = 0; i < 30; i++) {

            esamiTotali[i] = "esame n" + " " + (i+1);
        }

        PianoDiStudi pianodistudi = new PianoDiStudi(esamiTotali);

        Studente paperino = new Studente("Paperino", "Ingegneria", pianodistudi);
        System.out.println(paperino.getNome());
        System.out.println(paperino.getFacolta());
        System.out.println(paperino.getPiano());

        System.out.println("Paperino prende 25 al primo esame: \n\n");
        paperino.setEsame("esame n 1", 25);

        System.out.println("Paperino prende 30L al quinto esame: \n\n");
        paperino.setEsame("esame n 5", 31);
    }
}

```

```
System.out.print("Adesso Paperino ha la media del ");
System.out.println(paperino.media());
System.out.print("Questo perche' ha dato ");
    System.out.print(paperino.getNumEsamiFatti());
System.out.print(" in cui ha preso "); System.out.print(paperino.VotoEsame("esame
n 1"));
System.out.println(" e "); System.out.println(paperino.VotoEsame("esame n 5"));

System.out.println("Paperino cambia facolta' (ma i voti restano gli stessi):
\n\n");
paperino.setFacolta("Ingegneria dei Cupcake");
System.out.println(paperino.getNome());
System.out.println(paperino.getFacolta());
System.out.println(paperino.getPiano());

System.out.println("Quanto ha preso Paperino all'esame n. 25? ");
System.out.println(paperino.VotoEsame("esame n 25"));
}

}
```

24 Gestione delle Eccezioni in Java:

In un programma gli errori sono dovuti a:

- Errori di programmazione: il programmatore sbaglia qualcosa (divisione per zero, cast non permesso, accesso oltre i limiti di un array). **Questi errori non riguardano l'input e l'output**, e sono detti **unchecked**. Gli errori unchecked sono errori evitabili con la buona programmazione, e quindi non per forza da gestire.
- Errori di sistema: disco rotto, connessione remota chiusa, memoria non disponibile. **Questi errori riguardano comunque l'input e l'output**, e sono detti **checked**. Gli errori checked sono errori non evitabili con la buona programmazione, perché derivano dall'input/output e quindi non abbiamo potere di impedirli programmando bene. Quindi, nell'eventualità che si verifichino, vanno gestiti.
- Errori di utilizzo: input non corretti, tentativo di lavorare su file inesistente. **Anche questi errori sono dovuti all'input a all'output**, e quindi anche questi sono **checked**: per forza di cose devono essere gestiti.

Quindi gli errori di programmazione permetterebbero un programma senza input e output (quindi senza collegamento con l'esterno) che si riesce a scrivere senza dover gestire gli errori. Non appena comunico con l'esterno è molto più probabile dover gestire eccezioni, cioè nel caso di errori di sistema o di utilizzo.

24.1 Gerarchia di Classi in Java:

Parlamo di una **gerarchia di ereditarietà**: le eccezioni in Java sono in package diversi (ad esempio abbiamo java.lang e java.io) a seconda del tipo di errore. La superclasse di tutti gli errori è `Throwable` (che discende ovviamente da `Object` e che sta nel package.lang): quando definiamo noi delle nuove eccezioni anche loro sono figlie di `Throwable`. Su `java doc` su classe `Throwable` c'è scritto:

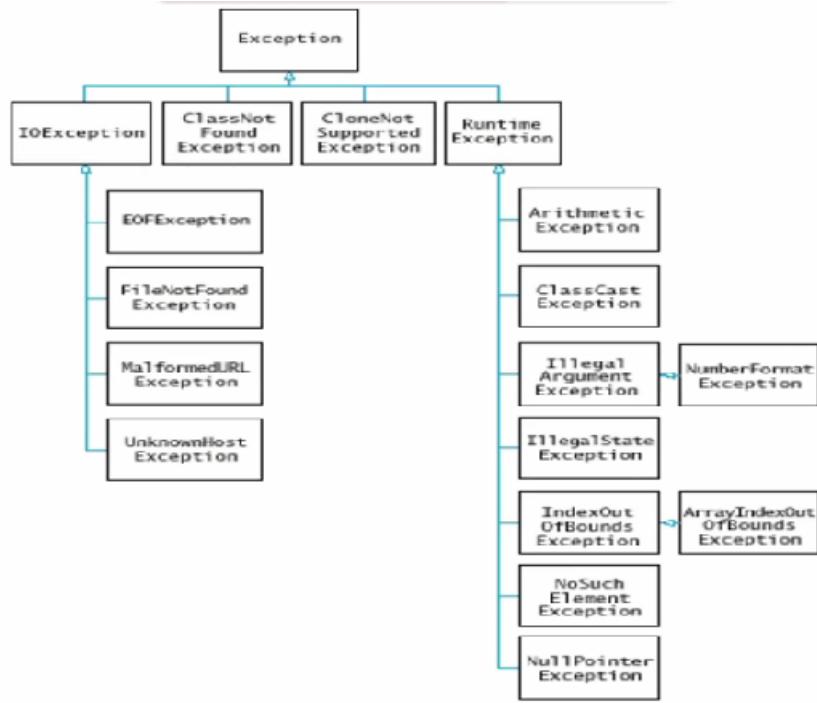
`Direct Known Subclasses: Error, Exception`

Queste sono le due sottoclassi dirette di `Throwable`, `Error` e `Exception`. Questo ci porta alla differenza concettuale tra un errore e un'eccezione:

- `Error`: contiene gli Errori fatali, cioè dovuti a condizioni incontrollabili. Quindi i programmi non gestiscono questi errori. Questa roba non c'entra con il discorso di prima: non sono né errori di input/output né errori di cattiva programmazione, è proprio roba greve tipo incompatibilità di versioni o esaurimento delle risorse necessarie alla JVM. Noi non avremmo a che fare con questi.
- `Exception`: tutti gli altri errori sono qui, e si chiamano eccezioni. Possono, come già detto, essere `caught` (e quindi li dobbiamo gestire) o `unchecked` (e non tocca gestirli). Un'eccezione termina un flusso di esecuzione indipendente del programma, cioè termina il thread. Quando si verifica un'eccezione il metodo trasferisce il controllo a un gestore delle eccezioni che decide cosa fare, ecco noi ci concentriamo su questa cosa adesso. **Tutte le classi di tipo eccezione sono ereditate dalla classe `Exception`**, proprio per la gerarchia di ereditarietà di cui parlavamo:

24.2 Eccezioni Controllate (checked) e Non Controllate (Unchecked):

Eccezioni Controllate: dovute a circostanze esterne che il programmatore non può evitare, quindi non prevedibili a tempo di scrittura del codice. Il compilatore però, nel caso in cui si verifichi l'eccezione, vuole sapere cosa fare, quindi richiede che queste eccezioni vengano gestite. Un es. `EOFException`: generata non solo per file, anche se abbiamo connessione di rete verso un client e il flusso di dati si blocca di colpo. Se un metodo ha una eccezione checked e non la gestisci il compilatore dà errore a tempo di compilazione. In generale tutte le sottoclassi di `IOException` sono eccezioni controllate (v. grafico sopra).



Eccezioni Non Controllate: dovute a circostanze esterne che il programmatore puo' evitare programmando bene. Es. `NullPointerException` o `IndexOutOfBoundsException`, che posso controllare programmaticamente. Un altro esempio di eccezione non controllata e' `NumberFormatException`, che si verifica quando `Integer.parseInt(str)` verifica che str non contiene un intero valido. Perche' questa eccezione e' non controllata? Il programmatore poteva farci qualcosa? Si'! Perche' ci sono metodi per controllare se una stringa e' un intero valido prima ancora di chiamare `Integer.parseInt`, quindi si poteva controllare a tempo di scrittura del codice. In generale tutte le sottoclassi di `RunTimeException` sono eccezioni non controllate (v. grafico sopra). Poi io potrei anche controllare le unchecked, se voglio farlo.

24.3 Ok ma, come gestire operativamente le eccezioni?

Bisogna catturare queste eccezioni per gestirle, altrimenti causano l'arresto del programma. Per **installare un gestore delle eccezioni**, di cui abbiamo gia' parlato, dobbiamo usare l'enunciato `try`, che avra' dentro il codice che vogliamo seguire, seguito da tante clausole `catch` quante sono le eccezioni che possono avvenire nel blocco `try` e che vogliamo o dobbiamo gestire. Quindi la struttura e' la seguente:

```

try
{
    enunciato
    enunciato
    ...
}

catch (ClasseEccezione oggettoEccezione) {

    enunciato
    enunciato
    ...
}

catch (ClasseEccezione oggettoEccezione) {

    enunciato
    enunciato
    ...
}

```

```
}

...
```

Osservazioni:

- oggettoEccezione e' una variabile il cui scope e' solo quello nel blocco del catch. oggettoEccezione da' informazioni sull'eccezione verificatasi, permettendomi di gestire l'eccezione.
- alla fine, dove stanno i puntini puo' esserci, opzionalmente, il blocco finally: il codice contenuto in questo blocco viene eseguito in ogni caso, sia che ci siano state eccezioni sia che non ci siano state. In realta', qualsiasi cosa dopo i catch viene eseguita, anche senza blocco finally, tranne in alcuni casi in cui il blocco finally e' fondamentale, ad esempio questi due motivi sono fondamentali per cui usare il blocco finally piuttosto che scrivere semplicemente dopo il catch:
 - Se nel blocco catch utilizzo un return, il programma si ferma perche' ritorna il valore del catch. Ma se io ho un blocco finally prima di terminare il programma esegue quel blocco finally, poi termina. Se io non usassi il blocco finally e semplicemente scrivessi dopo il catch, se uso il return quello che c'e' dopo il catch viene ignorato
 - Quando sono nel finally io so che il codice e' in un determinato stato, mentre se non lo uso dopo il catch non so il codice in che stato e'.

24.3.1 Dettagli sulla classe Exception:

e' una classe derivata di Throwable, lo avevamo gia' detto, ma inoltre:

- Ha due costruttori:
 - Exception() che costruisce un'eccezione senza uno specifico messaggio
 - Exception(String msg) che costruisce un'eccezione con il messaggio msg
- Eredita da Throwable il metodo String getMessage() che restituisce come stringa il messaggio contenuto nell'eccezione
- Possiamo anche creare l'eccezione nel momento in cui viene lanciata, non solo prima.
- Il Garbage Collector distrugge solo le eccezioni che abbiamo catturato, cioe' per cui e' stato eseguito il blocco catch.

24.4 Un bell'esempio conclusivo:

```
import java.io.*;

class ClasseEcc{

    void ff() throws IOException {

        Exception ec = new Exception("Prima eccezione");
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        int i = Integer.parseInt(br.readLine());

        try {
            if (i == 1) throw ec;
            throw new Exception("Altra eccezione");
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }

        System.out.println("Proseguione della funzione");
    }
}
```

```

public static void main (String[] args) throws IOException
{
    ClasseEcc ex = new ClasseEcc();
    ex.ff();
    System.out.println("Proseguzione del programma");
}

```

Osservazioni:

- Vediamo che il metodo ff() e' seguito da throws IOException: sto dichiarando che il mio metodo ff() puo' lanciare delle IOException, che sono catched. Quindi solo le istruzioni catched devono obbligatoriamente palesarsi nel throws, le eccezioni uncatched non per forza, vedremo un esempio in un prossimo punto. Questo e' un altro modo di *gestire le eccezioni*.
- Ma il metodo ff() che fa? Vediamo dopo la prima istruzione. La seconda istruzione crea un'istanza di InputStreamReader (che e' una classe che permette di leggere da un flusso dati). Noi fino ad adesso per leggere da System.in abbiamo usato Scanner (che e' una facility di Java), dietro Scanner ci sono tutta una serie di classi che poi vedremo. Qui abbiamo usato un altro modo per leggere da System.in, che e' appunto creare un'istanza di InputStreamReader poi creare un'istanza di BufferedReader, che invece fornisce dei metodi per leggere riga per riga, come appunto readLine() che andiamo ad utilizzare subito dopo. Di readLine() **andrebbe gestita l'eccezione**, perche' questa rilancia (con il throws anche lei) un'eccezione. Infatti se andiamo a cercarla sulla java doc troviamo:

```
public String readLine() throws IOException
```

Noi che facciamo? Non la gestiamo con try e catch, **ma la rilanciamo**, facendo:

```
void ff() throws IOException
```

Che significa che la rilancio cosi'? Che chi utilizzerà ff() dovrà gestire adesso lui la IOException, usando try e catch, oppure rilanciandola a qualcun'altro con throws anche lui.

Nota che con throws si possono rilanciare anche eccezioni unchecked, ma proprio perche' sono unchecked, non saremo costretti a gestirle.

- Exception ec = new Exception("Prima eccezione"); e' l'istruzione che *crea preventivamente, prima di lanciarla*²⁶, l'istruzione ec. **IMPORTANTE!** Vediamo che ec e' di tipo Exception, ma dopo void ff() throws abbiamo soltanto IOException, e non la classe Exception: cio' significa che ec, che e' della classe Exception, e' una eccezione uncatched; cioe' non deve essere gestita da chi usa ff(). L'eccezione uncatched ec viene lanciata poi dopo, nel blocco try, nel caso in cui i == 1. Se invece i e' diverso da 1 eseguo l'altra istruzione, cioe':

```
throw new Exception("Altra eccezione");
```

In questo caso invece l'eccezione viene creata nello stesso momento in cui viene lanciata, diversamente dalla prima eccezione che era stata creata preventivamente.

- Il blocco try quindi ha due possibili eccezioni, e ogni volta si verifica una e una sola delle due eccezioni in base al valore di i **questo perche' la prima eccezione che viene lanciata con throw fa uscire dal blocco try e entrare nel blocco catch**. Ogni volta quindi viene lanciata una eccezione e quindi ogni volta viene attivato il blocco catch sotto: se viene lanciata la prima eccezione il parametro e di catch e' relativo alla prima eccezione, e quindi getMessage() ci da' "Prima eccezione". Invece se viene lanciata la seconda eccezione il parametro e di catch e' relativo alla seconda eccezione e quindi getMessage() ci da' "Altra eccezione".

²⁶Abbastanza rara come cosa, generalmente le eccezioni vengono istanziate nel momento in cui sono lanciate, come vediamo tra poco.

- Notare che abbiamo un metodo ff() e un main nella stessa classe, non e' una cosa pulita, ma si fa quando si vuole essere rapidi nel programmare.

Quindi abbiamo anche visto come generare le eccezioni noi, non quelle di default ma crearle noi.

Nota: il parseInt() non e' nel blocco try. Cio' significa che se l'utente non inserisce un numero, il programma si blocca!! Questo perche' parseInt fuori dal blocco try significa che non stiamo gestendo l'eccezione NumberFormatException. Il discorso e' che e' un'eccezione non controllata, quindi non e' che va gestita per forza con try e catch: ci sono dei metodi che ci permettono di verificare se una stringa possiamo renderla intero, quindi e' un'eccezione che possiamo controllare a tempo di scrittura del codice con buona programmazione. Il problema e' che non stiamo facendo ne' l'uno nell'altro: non stiamo ne' usando questi metodi per controllare a tempo di scrittura del codice, ne' inserito il parseInt nel blocco try, in modo da gestire l'eccezione uncaught come fosse caught, dato che comunque possiamo farlo nessuno ce lo vieta: semplicemente non e' obbligatorio.

Che succede se non inserisco un valore stringa che puo' essere convertito correttamente in intero in br.readLine()? Tipo se inserisco "gh"? Succede che parseInt genera l'errore NumberFormatException e lo propaga al main. In console comparira' il seguente messaggio di errore:

```
gh
Exception in thread "main" java.lang.NumberFormatException: For input string: "gh"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at Prova.ff(Prova.java:21)
    at Prova.main(Prova.java:35)
C:\Users\Francesco Letta\AppData\Local\NetBeans\Cache\12.5\executor-snippets\run.xml:111: The following error occurred while executing this line:
C:\Users\Francesco Letta\AppData\Local\NetBeans\Cache\12.5\executor-snippets\run.xml:94: Java returned: 1
BUILD FAILED (total time: 28 seconds)
```

Analizziamo partendo dalla prima riga questo messaggio:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "gh"
```

Ci dice che si e' verificata un'eccezione nel thread (spiegheremo piu' avanti cos'e' il thread!!!) main. Questa eccezione e' di tipo NumberFormatException, che e' una classe che si trova nel package java.lang. L'errore si e' verificato per aver inserito in input la stringa "gh".

```
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
```

Questo e' dove l'errore si e' verificato precisamente: a riga 65 del file NumberFormatException.java. Ma che e' forInputString?? Si tratta di un metodo chiamato da parseInt, come vediamo sotto, quindi la chiamata di forInputString fa parte dell'implementazione di parseInt. Vediamo che tra parentesi ci dice che forInputString e' un metodo della classe NumberFormatException (cosa che in realta' ci veniva detta anche a inizio riga, con l'aggiunta del package di appartenenza java.lang), presente nel file NumberFormatException.java.

```
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
```

L'errore da forInputString si **propaga** alla chiamata di parseInt, appartenente alla classe Integer di java.lang. Poi questo viene propagato a un'altro parseInt, che vuol dire? Si tratta di overloading: si vede che il secondo parseInt, quello che abbiamo chiamato noi, fa overloading di un parseInt originale, e lo chiama quest'ultimo nel suo corpo, come si fa di solito. Quindi per ora sappiamo questo: il nostro parseInt, che e' overload del parseInt originale, chiama quest'ultimo, e il parseInt originale chiama forInputString. forInputString genera l'errore e lo propaga al parseInt originale, che lo propaga al parseInt overload che abbiamo chiamato noi. Andiamo avanti:

```
    at Prova.ff(Prova.java:21)
```

Il nostro parseInt era chiamato dal nostro metodo ff() - che si trova nella nostra classe Prova - , quindi parseInt propaga alla chiamata di ff() l'eccezione, precisamente a riga 21 del file.

```
    at Prova.main(Prova.java:35)
```

Il nostro metodo f() propaga al main della classe Prova l'eccezione, e qua si ferma.

Quindi tramite questa **lettura dello stack trace** riusciamo a capire qual e' la sequenza di chiamate che ha generato l'eccezione. Ok, ma allora proviamo a sistemare sta eccezione unchecked che si verifica!

```

...
try {
    int i = Integer.parseInt(br.readLine());
} catch (NumberFormatException e) {
    ...
}

try {
    if (i == 1) throw ec; //ERRORE: i non e' visibile qua
    throw new Exception("Altra eccezione");
}
...

```

Notiamo che cosi' ci da' errore! Perche' `int i = Integer.parseInt(br.readLine());` e' all'intero del blocco try, quindi ho dichiarato una variabile nel blocco try. Sappiamo che le variabili che dichiariamo nel blocco try sono visibili solo in questo blocco, quindi lo scope della variabile `int i` sara' solo il blocco try in cui e' stata definita. Questo significa che nel successivo blocco try, l'istruzione `if (i == 1)` non avra' senso, perche' `i` non e' una variabile visibile fuori dal suo blocco try, quindi non esiste. Per sistemarlo, o usiamo un solo blocco try oppure se vogliamo usarne di piu' facciamo cosi':

```

import java.io.*;

class ClasseEcc{

    void ff() throws IOException {

        Exception ec = new Exception("Prima eccezione");
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);

        int i = 0;
        try {
            i = Integer.parseInt(br.readLine());
        } catch (NumberFormatException e) {
            System.out.println("Non un numero!!!")
            return;
        }

        try {
            if (i == 1) throw ec;
            throw new Exception("Altra eccezione");
        }
        catch(Exception e) {
            System.out.println(e.getMessage());
        }

        System.out.println("Proseguire della funzione");
    }

    public static void main (String[] args) throws IOException
    {
        ClasseEcc ex = new ClasseEcc();
        ex.ff();
        System.out.println("Proseguire del programma");
    }
}
```

In questo modo fermiamo il programma, un'altra cosa che si poteva fare e' far continuare il programma, proponendo di nuovo l'input nel caso in cui se ne inserisca uno errato. Ad esempio cosi', con un ciclo while:

```
...
boolean corretto = false;

while (!corretto) {
    try {
        i = Integer.parseInt(br.readLine());
        corretto = true;
    } catch (NumberFormatException e) {
        System.out.println("Non un numero!!!!");
    }
}
```

24.5 Progettare nuove eccezioni:

Generalmente le eccezioni di Java sono esaustive e vanno bene per tutto cio' che ci serve, ma se non dovesse essere cosi', possiamo progettare noi nuove eccezioni e inserirle obbligatoriamente nella discendenza di Throwable. Generalmente, ma non e' obbligatorio, si inseriscono precisamente come sottoclassi di RunTimeException, quindi eccezioni unchecked. Ok, ma come progettarla? Introduciamo un nuovo tipo di eccezione con una classe, che sara' sottoclasse sicuramente di Throwable, probabilmente di RuntimeException, se vogliamo di una sottoclasse di RunTimeException (es. possiamo estendere IllegalArgumentException). Facciamo un esempio:

```
public class DivisionePerZeroException extends RuntimeException {

    public DivisionePerZeroException() { //costruttore vuoto
        super("Divisione per zero!");
    }

    public DivisionePerZeroException(String msg) { //costruttore con messaggio
        super(msg);
    }
}
```

Questi due costruttori vanno messi per forza: il costruttore vuoto va messo esplicitamente e con, nel corpo, una chiamata al costruttore con parametro della superclasse RunTimeException. Perche'? Perche' la RunTimeException non ha un costruttore vuoto, quindi se io non esplicitassi un costruttore vuoto in DivisionePerZeroException poi il costruttore di default di quest'ultimo cercherebbe di chiamare il costruttore vuoto di RunTimeException, che pero' non c'e'. Quindi che fare? Dobbiamo fare il nostro costruttore vuoto in DivisionePerZeroException e dobbiamo fargli chiamare il costruttore con un parametro della superclasse RunTimeException. Oltre a questi due costruttori obbligatori, optionalmente possiamo aggiungere eccezioni piu' ricche, con piu' informazioni, come variabili o metodi. Bene, scriviamo un metodo e un main per provare questa nuova eccezione:

```
public class Divisione {

    public Divisione(int n, int d) {
        num=n;
        den=d;
    }

    public double dividi(){
        if (den == 0)
            throw new DivisionePerZeroException(); //invoca il costruttore senza parametri
        return num/den;
    }

    private int num;
```

```

    private int den;
}

-----
public class Test {

    public static void main(String[] args) throws IOException {

        double res;
        Scanner scan = new Scanner(System.in);
        System.out.print("Inserisci il numeratore:");
        int n = Integer.parseInt(scan.nextLine());
        System.out.print("Inserisci il denominatore:");
        int d = Integer.parseInt(scan.readLine());

        try {
            Divisione div = new Divisione(n,d);
            res = div.dividi();
            System.out.print(res);
        } catch(DivisionePerZeroException exception) {
            System.out.println(exception);
        }
    }
}

```

Osservazione: invece di stampare un messaggio stringa abbiamo stampato un'istanza della classe DivisionePerZeroException. Quindi stiamo utilizzando un `toString()`. Precisamente, dato che non abbiamo fatto nessuna definizione di `toString()` in `DivisionePerZeroException`, si tratta del `toString()` di `RuntimeException`, che ovviamente override quello di `Object`. Questo `toString()` e' stato ridefinito per descrivere l'oggetto exception con una stringa costituita da:

- Nome della classe a cui l'oggetto exception appartiene, nel nostro caso `DivisionePerZeroException`, seguito da:
- il carattere " . ", seguito da:
- il messaggio di errore associato dal costruttore all'oggetto exception, nel nostro caso si tratta di "Divisione per zero!".

24.6 Catturare eccezioni:

se non catturiamo l'eccezione ci viene una scritta rossa con stack trace. L'inconveniente di questo e' che muore il thread, e poiche' per ora facciamo solo programmi monothread questo significa che si ferma l'intero programma. Se non vogliamo far morire il thread ma comunque stampare la stack trace utilizziamo il metodo `printStackTrace()`. Quindi in un blocco catch possiamo fare cosi':

```

...
catch (DivisionePerZeroException exception) {

    exception.printStackTrace();
}

```

24.6.1 La clausola Finally:

Facciamo innanzitutto un'osservazione: e' impossibile entrare in piu' blocchi catch relativi a un singolo blocco try nello stessa esecuzione. Ogni blocco catch puo' essere relativo a un'eccezione specifica del blocco try, ma una volta che si verifica la prima eccezione del blocco try, e quindi si entra nel blocco catch ad essa collegata, non si rientra nel blocco try, quindi non si verificano altre sue eccezioni. Ok ora il finally, viene eseguito dopo il catch che viene eseguito:

```
try {
    ...
} catch {
    ...
}
finally {
    ...
}
```

Nel finally si fanno operazioni di pulizia per chiudere.

Part VI

Java Collections Framework:

Un framework e' un sinonimo di libreria. Quindi le JCF sono le librerie Java che si occupano delle collezioni. Le collezioni sono raccolte di instance o di valori: es. insiemi, linked lists, mappe (dizionari) ecc. . Tutte le collezioni sono definite nelle librerie del JCF.

25 Tipi Generici (Generics):

Vengono usati in Java per definire classi, interfacce e metodi **parametrici** rispetto ai tipi di dati su cui operano, cioe'? Cioe' utilizzo un tipo generico se voglio costruire una classe, un'interfaccia o un metodo che non voglio vincolare a lavorare su uno specifico tipo (es. classe che definisce liste, non voglio vincolarla a liste di stringhe, ma di qualsiasi tipo). Prima di Java 5 non c'erano i Generics, come si faceva? Due modi:

- Si definiva la classe, l'interfaccia o il metodo in questione una volta per uno specifico tipo, poi lo si copiava e incollava cambiando il tipo che serviva, per ogni tipo che si voleva utilizzare. Esempio, se volevamo costruire una Coppia, allora prima facevamo:

```
public class CoppiaDiInteri {
    private int primo;
    private int secondo;

    public CoppiaDiInteri(int a, int b) {
        primo = a; secondo = b;
    }
    public int getPrimo() {
        return primo;
    }
    public int getSecondo() {
        return secondo;
    }
}
```

E poi copiavo e incollavo il codice per ogni altro tipo per cui mi serviva, esempio:

```
public class CoppiaDiStringhe {
    private String primo;
    private String secondo;

    public CoppiaDiStringhe(String a, String b) {
```

```

        primo = a;
        secondo = b;
    }
    public String getPrimo() {
        return primo;
    }
    public String getSecondo() {
        return secondo;
    }
}

```

- Si costruiva la classe, in modo che operasse sul tipo piu' generico possibile, che sappiamo essere il tipo Object, in modo da poter SEMPRE castare a qualsiasi tipo (perche' ogni classe e' figlia della classe Object). Ad esempio:

```

public class Coppia {
    private Object primo;
    private Object secondo;

    public Coppia(Object a, Object b) {
        primo = a; secondo = b;
    }
    public Object getPrimo() {
        return primo;
    }
    public Object getSecondo() {
        return secondo;
    }
}

```

E quindi un Client di Coppia avrebbe dovuto fare:

```

...
Coppia c = new Coppia("ciao", "mondo");

String primo = (String)c.getPrimo();

String secondo = (String)c.getSecondo();

...

```

Quindi c'era la necessita' di fare ogni singola volta un cast. Inoltre nota che questo non era compatibile con l'utilizzo di dati primitivi, quali int, double ecc., poiche' non essendo classi, non esiste il casting a questi. La soluzione a questo pero' erano le cosiddette *wrapper class*, cioe' le classi Integer, Double ecc., che prendevano i tipi di dato primitivi e li rendevano delle classi, rendendo possibile il casting.

Quindi il discorso e' che si cercava un modo per scrivere una sola classe, interfaccia o metodo, invece di tante diverse, che condividessero la stessa logica in tutto e per tutto, e che differissero solo per il tipo su cui operavano. Oggi per questo scopo si utilizzano le Generics, ossia i Tipi Generici. Vediamo subito un esempio:

```

public class Coppia<T> {
    private T primo;
    private T secondo;

    public Coppia(T a, T b) {
        primo = a; secondo = b;
    }
    public T getPrimo() {
        return primo;
    }
}

```

```

    public T getSecondo() {
        return secondo;
    }
}

```

Quindi aggiungiamo T_i accanto al nome della classe. T e' il nome del **parametro di tipo formale**, che e' solo un *placeholder*, perche' poi quando andiamo a istanziare Coppia:

```

Coppia<String> c = new Coppia<String>("ciao", "mondo");
String primo = c.getPrimo();
String secondo = c.getSecondo();

```

Il tipo parametrico T e' stato sostituito con un tipo concreto, detto **parametro di tipo attuale**. Si possono utilizzare piu' tipi inserendo la lettera o la stringa nella stessa parentesi angolare ma separati da virgola. Ad esempio nei dizionari si utilizza un tipo per il valore e un tipo per la chiave quindi si fa cosi'.

Convenzione sui nomi dei parametri di tipo formale: si puo' utilizzare per i loro nomi qualsiasi lettera o stringa, tuttavia e' convenzione usare queste lettere:

- E, per gli elementi di una collezione
- K, per le chiavi di un dizionario
- V, per i valori di un dizionario
- N, per i numeri
- T, per i tipi, cioe' per le classi (come abbiamo fatto sopra per String, o che ne so con Persona, insomma con le altre classi)
- S,U,V per ulteriori parametri oltre il primo nelle parentesi angolate.

Pero' sono solo convenzioni.

25.1 Tipi Generici e Sottoclassi:

Possiamo fare due cose:

- come a un riferimento di tipo C possiamo assegnare istanze di un tipo D , nel caso in cui D sia sottoclasse di C , e' anche vero che a un riferimento di tipo generico C_iT_i possiamo assegnare istanze di un generico tipo D_iT_i , nel caso in cui D_iT_i sia sottoclasse di C_iT_i . Ad esempio:

```

class Pila<E> {

    public void inserisci(E elem) {...} //inserisce nella pila
    public E rimuovi() {...} //rimuove dalla pila e restituisce

}

class PilaCollegata<E> extends Pila<E> {...}

```

Cosicche' un eventuale client possa fare:

```

Pila<Number> pn = new PilaCollegata<Number>();

```

Dove abbiamo utilizzato come parametro di tipo attuale il tipo Number, che e' la superclasse di tutte le wrapper class relative ai numeri in Java (quindi Integer, Double ecc.). Vediamo che il nome del parametro di tipo formale di Pila e PilaCollegata e' E perche' E sono gli elementi di una collezione, collezione che e' appunto la pila.

- Dato un tipo generico C_iT_i , possiamo, in fase di invocazione, utilizzare una qualsiasi sottoclasse di T , riprendiamo la classe generica Pila e la classe Number, che abbiamo detto essere superclasse di tutte le wrapper class, per fare un esempio:

```
Pila<Number> pn = new Pila<Number>();
pn.inserisci(new Integer(1));
pn.inserisci(new Double(0.5));
```

Ma non possiamo fare questa cosa:

- Non possiamo assegnare a un'istanza di un tipo generico $C_i T_j$ un'istanza dello stesso tipo generico ma con diverso parametro attuale. Ad esempio:

```
Pila<Number> pn = new Pila<Integer>(); //ERRORE
```

Ci verrà generato un errore a tempo di compilazione, perché? Perché se questo fosse possibile io potrei fare cose così:

```
pi = new Pila<Integer>();

Pila<Number> pn = new Pila<Integer>(); //Facciamo finta non sia errore e che
                                         avvenga una copia superficiale, quindi pn ora punta alla stessa pila di pi,
                                         anche se e' un riferimento a una pila di Number.

pn.inserisci(new Double(0.5));

Integer i = pi.rimuovi() //NON SI PUO FARE, cerca di assegnare un double
                         (perché pi punta alla stessa pila di pn, in cui prima era stato inserito
                          uno double), ad un riferimento di tipo Integer, ma essendo due classi
                          diverse, non figlie una dell'altra, questa cosa non si può fare
```

WildCards: Esiste un tipo generico speciale $C_i ? T_j$, il cui parametro è sconosciuto: con questo saremo in grado di ottenere degli oggetti dalla pila (con il metodo `rimuovi()`) ma non di aggiungerli (con il metodo `inserisci()`) poiché il tipo `?` è sconosciuto.

25.2 Cancellazione (Erasure):

I tipi generici esistono per il tempo di scrittura del codice: a tempo di compilazione ogni occorrenza del parametro di tipo formale viene sostituito con un tipo `Object`, poi castato esplicitamente dove richiesto ai vari parametri di tipo attuale. Questo spiega anche il motivo per cui i parametri di tipo attuale non possono essere tipi primitivi: a tempo di compilazione devo fare il cast, come faccio a castare `Object` in un tipo primitivo. Come abbiamo già detto se vogliamo usare i tipi primitivi usiamo le wrapper class.

26 Java Collections Framework:

Ok, fino ad ora abbiamo parlato di cose propedeutiche a JCF, adesso parliamo proprio di questo. Libreria di interfacce e di classi che implementano queste interfacce. Queste interfacce e queste classi ci permettono di lavorare con collezioni di oggetti. Precisamente la JCF è formata da:

- Interfacce: sono le categorie di collezioni. Queste categorie sono quindi lista (permettono la ripetizione e sono ordinati), insieme (non permettono la ripetizione e non sono ordinati) e mappe (cioè i dizionari).
- Implementazioni: sono le classi che implementano le interfacce. Per farlo utilizzano strutture dati efficienti. Ci sono differenti implementazioni per ogni categoria di collezione.
- Algoritmi: si tratta di funzioni che realizzano algoritmi di ricerca e di ordinamento sugli oggetti delle classi concrete che implementano le interfacce. Esempi: ricerca binaria;; quick sort.

Perché utilizzare la JCF piuttosto che crearmi io la mia collezione? Ad esempio, perché dovrei utilizzare la linked list di JCF al posto di scrivermi io la mia classe linked list e usarla, come abbiamo sempre fatto? I vantaggi nell'utilizzo della JCF sono:

- Generata': data dalle interfacce. Posso decidere infatti di scrivere una mia implementazione per una collezione specifica (ad esempio una lista ordinata in modo strano), perche' so che questa implementazione si comporta meglio sul mio data set specifico. Quindi grazie alla JCF possiamo modificare l'implementazione di una collezione senza dover modificare i Client: se ho un codice che funzionava con l'implementazione x della lista, se nella JFC ho anche un'altra implementazione y della lista, posso cambiare implementazione ma il codice resta lo stesso e non devo cambiarlo.
- Interoperabilita': la JFC e' stata costruita da qualcun'altro, ma a me non interessa, io scrivo codice per usare la JCF indipendentemente dal codice con cui questa e' stata costruita.
- Efficienza: le classi concrete che realizzano la collezione sono ottimizzate per fare quello con buone prestazioni

Tipi Generici: Nella libreria JCF vengono usati i generics per le collezioni di oggetti. In questa maniera queste collezioni di oggetti diventano di tipo parametrico e l'utente puo' facilmente cambiare il tipo della sua collezione. Il vantaggio di usare i tipi generici invece che la classe Object, oltre a quello di non dover castare manualmente ogni volta, e' quello di avere un controllo sulla correttezza dei tipi di dato a tempo di compilazione e non a tempo di esecuzione come nel caso di Object. Nel caso di Object infatti sappiamo che castando ad un certo tipo il compilatore si fida e ci lascia fare, e eventuali problemi per un tipo non compatibile si rivelano solo a tempo di esecuzione. Con i generics no invece, a tempo di compilazione possiamo verificare bene. Inoltre con i generics impediamo la promiscuita' dei dati. Ad esempio, prendendo una collezione di JCF con generics di tipo Persona (cioe' ad esempio Collection<Persona>), saremo sicuri a tempo di compilazione di non poter avere all'interno della collezione altri tipi di dato, che non siano Persona o sue sottoclassi, impedendo quindi l'ingresso di dati incompatibili con Persona.

Sia le interfacce che le classi del JCF sono definite nella libreria `java.util`, quindi quando dovremo usarle dobbiamo importare il package. Ecco la gerarchia per l'interfaccia Collection: Vediamo che

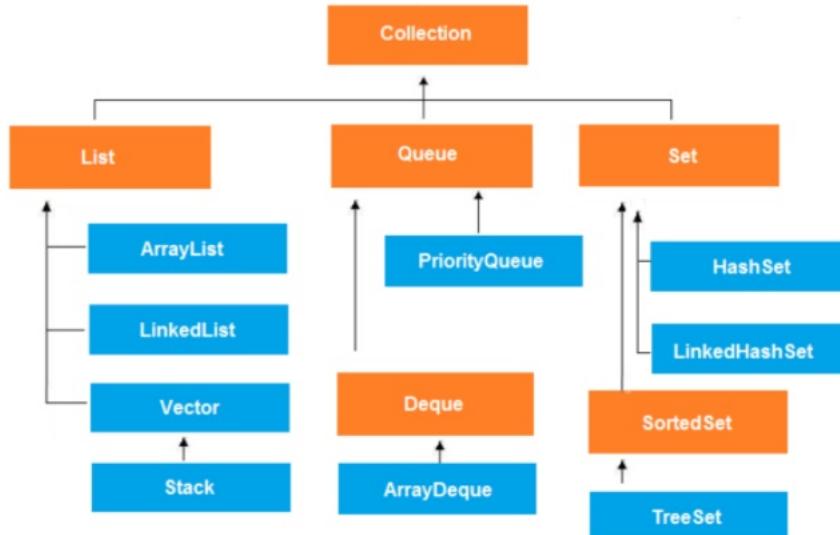


Figure 10: Interfacce derivate di Collection e classi che implementano. In arancione le interfacce in blu le classi.

l'interfaccia List ha le seguenti implementazioni:

- ArrayList: lista fatta usando array dinamici
- LinkedList: lista implementata con liste collegate
- Vector: lista fatta utilizzando la classe Vector di Java. Sottoclasse di Vector e' la classe Stack, che e' un'implementazione di List che segue la politica LIFO (struttura dati utilizzata per l'albero delle chiamate). La Vector e' usata molto meno rispetto ad ArrayList e LinkedList.

Per quanto riguarda l'interfaccia Queue invece, abbiamo le seguenti implementazioni:

- PriorityQueue: una coda di priorità dove vengono estratti elementi seguendo il minimo.
- Per quanto riguarda l'interfaccia Deque abbiamo le seguenti implementazioni:
 - ArrayDeque: fatta con l'array.

Per quanto riguarda l'interfaccia Set invece abbiamo come implementazioni:

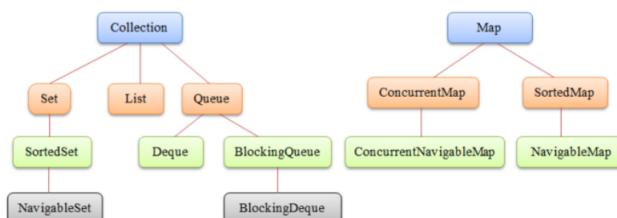
- HashSet: si basa sull'hashCode. Fa uso della funzione hash applicata all'oggetto.
- LinkedHashSet: deriva dall'HashSet ma usa una struttura collegata
- Per quanto riguarda l'interfaccia SortedSet:
 - TreeSet, implementazione fatta con un albero di ricerca bilanciato, che quindi permette ricerca facile e ordinamento veloce.

Adesso concentriamoci sulle interfacce.

26.1 Interfacce del JCF:

Quelle che ci interessano per il corso sono:

- Collezioni
 - Insiemi: collezione di dati non ordinati e senza ripetizione. I SortedSet sono insiemi ordinati.
 - Liste: collezione di elementi ordinati
 - Code: politica FIFO, utilizzate per i buffer. Che cos'è un buffer: se abbiamo un qualcosa che processa dati, e arrivano in ingresso più dati di quelli che possiamo processare, un buffer (nel nostro caso una coda) bufferizza i messaggi, così da evitare di perdere messaggi in ingresso. Un tipo di coda è la Deque: una coda a doppia uscita, cioè da entrambi i versi escono e entrano dati. La BlockingQueue invece gestisce le relazioni in concorrenza.
- Mappe (dizionari, v. dopo): insiemi di coppie `{chiave, valore}`. Essendo insiemi per definizione non sono ordinati.
 - ConcurrentHashMap: gestione delle operazioni in concorrenza
 - Sorted Map: dizionari ordinati rispetto alle chiavi



Bisogna sapere che nel JCF **due oggetti sono uguali se sia equals() e' true sia hashCode e' true()**. Ovviamente questo vale anche fuori dal JCF

26.1.1 Interfaccia Collection:

Concentriamoci sull'interfaccia Collection, ecco il suo codice:

```

public interface Collection<E> {

    // Basic Operations
    int size();
    boolean isEmpty();
  
```

```

boolean contains(Object element); // prende Object e non E perche' contains usa
    equals(), e noi potremmo avere un equals() particolarmente intelligente che
    confronta anche istanze di due tipi diversi; ad esempio equals() che confronta
    Persona (che ha vari campi dato, tra cui il campo String nome) con una stringa:
    equals() lo costruiamo in modo che confronti il nome della persona con l'altra
    stringa. Quindi nel dubbio mettiamo a Object

boolean add(E element); // Optional (ci sono delle implementazioni che sono
    immutabili). add() prova a inserire un elemento e restituisce true se l'elemento
    viene inserito e false se non viene inserito (es. provi a inserire un elemento
    gia' presente in un insieme).

boolean remove(Object element); // Optional (ci sono delle implementazioni che sono
    immutabili, quindi li' non vanno). remove() prova a rimuovere un elemento e
    restituisce true se l'elemento viene rimosso e false se non viene rimosso (es.
    provi a rimuovere un elemento non presente in un insieme).

Iterator<E> iterator(); // v. dopo, molto legato al ciclo foreach

boolean equals(Object o);

// Bulk Operations, cioe' operazioni su un blocco di dati
boolean containsAll(Collection<?> c); // Restituisce true se tutti gli elementi di c
    sono nella collezione di invocazione. Vediamo che containsAll prende in ingresso
    una Collection di ?, quindi utilizziamo le wildcards, perche'? Vediamo dalle
    Basic Operation che contains prende in input un parametro di tipo Object, per i
    motivi che abbiamo detto sopra, allora containsAll dovrebbe prendere in input una
    collezione di Object: possiamo quindi scrivere Collection<Object> o piu'
    brevemente Collection<?>.

boolean addAll(Collection<? extends E> c); // Optional. Prende in ingresso una
    collection di QUALUNQUE CLASSE CHE ESTENDE E. Questo e' un utilizzo molto comune
    delle wildcards. Vediamo che <? extends E> e' molto diverso da <T extends E>; in
    quest'ultimo caso stiamo dicendo al programmatore di andare a specificare una
    precisa classe T che estenda E quando va nei client (es. main) a creare la
    collezione istanza di Collection; invece con <? extends E> permettiamo al
    programmatore di inserire nella collezione istanza di Collection che crea dati di
    qualunque tipo che siano pero' sottoclasse di E.

boolean removeAll(Collection<?> c); // Optional. Eliminiamo tutti gli elementi di c
    dalla collezione oggetto di invocazione

boolean retainAll(Collection<?> c); // Optional. Fa il contrario di removeAll,
    elimina tutti gli elementi dalla collezione oggetto di invocazione tranne quelli
    presenti anche in c

void clear(); // Optional. Elimina proprio tutto tutto cio' che c'e' nella collezione

// Array Operations
Object[] toArray(); // restituisce un'array di Object
<T> T[] toArray(T[] a); // Restituisce un'array di tipo T. Perche', oltre a
    restituire un array di tipo T, ne prende pure in input uno? Si tratta di un
    trucco per aggirare i limiti del linguaggio Java: se io faccio un metodo
    generics, il tipo del generics deve essere utilizzato anche in uno degli
    argomenti del metodo, quindi passiamo un array vuoto del tipo T come parametro al
    metodo
}

```

Nota su Optional: quando un metodo e' Optional vuol dire che solitamente li possiamo implementare lanciando un'eccezione, se in una classe che implementa l'interfaccia non vogliamo utilizzare quel metodo. Nella JavaDoc e' spiegato tutto bene. Prendiamo nella JavaDoc l'interfaccia ListIterator;E, vediamo il metodo add per esempio, che e' appunto Optional. Vediamo che add in fondo ha scritto:

`UnsupportedOperationException` - if the add method is not supported by this list iterator

Quindi va implementato per forza, se non lo vuoi implementare dobbiamo lanciare quell'eccezione.

Vediamo Set e List, due interfacce derivate dell'interfaccia Collection.

```
public interface Set<E> extends Collection<E> {  
}
```

Vediamo che Set non definisce nulla di nuovo rispetto a Collection, quindi eredita tutti i suoi metodi ma non ha nient'altro. Ma allora perche' creare Set? Perche' quando io voglio *marcare* delle classi decido di costruirmi delle interfacce apposite. Cioe' in questo caso sto dicendo: l'insieme si comporta in una determinata, peculiare, maniera, ma i metodi che utilizza sono gli stessi di Collection; allora definisco una interfaccia che non ha metodi aggiuntivi: in questo modo basta invocare `instanceOf` Set su un'istanza per sapere se e' un insieme, e quindi se si comporta come un insieme, e in questo modo ho *marcato* quell'istanza e ho capito chi e'. Vediamo l'interfaccia List:

```
public interface List<E> extends Collection<E> {  
  
    boolean add(int index, E element); // Optional (per gli stessi motivi). Siccome la  
        lista e' una collezione posizionale l'add specializzato di List ci permette di  
        specificare una posizione. Se non do in input la posizione Java mi fa utilizzare  
        l'add di Collection e, a seconda dell'implementazione, l'elemento verrà messo da  
        qualche parte (ci potrebbe essere l'implementazione che mette in coda o in testa  
        ecc.).  
  
    E get(int index); //restituisce l'elemento in posizione index nella lista  
  
    E set(int index, E element); // Optional (anche qui, ci sono classi immutabili e  
        quindi per quelle questo metodo non c'e'). Modifica l'elemento in posizione index  
        con element  
  
    int indexOf(Object o); //restituisce posizione prima occorrenza di o  
  
    int lastIndexOf(Object o); //restituisce posizione ultima occorrenza di o  
  
    boolean remove(int index); //rimuove elemento in posizione index  
  
    ListIterator<E> listIterator(); //restituisce un iterator particolare: un list  
        iterator, che ha delle funzionalità peculiari  
  
    ListIterator<E> listIterator(int index) //restituisce un list iterator facendolo  
        partire dalla posizione index  
  
    boolean addAll(int index, Collection<? extends E> c); // Optional (per gli stessi  
        motivi). Aggiunge tutti gli elementi di c nella lista, a partire dall'indice index  
  
    List<E> subList(int fromIndex, int toIndex); //estrae e restituisce una sottolista,  
        da fromIndex a toIndex  
}
```

Solitamente la list ha implementazione con side-effect e senza condivisione di memoria. In realta' sugli oggetti specifici c'e' condivisione ma non sulle classi che compongono la struttura dati. Cioe' c'e' il cosiddetto **SHARING**, che abbiamo gia' visto, cio' vuol dire che gli oggetti di tipo E sono in condivisione di memoria, mentre gli oggetti di tipo List<E> non sono in condivisione di memoria.

26.1.2 Iteratori:

Un iteratore e' un cursore con cui si scandisce una collezione. Ogni iteratore e' associato a una collezione. Un'iteratore e' precisamente un'istanza di una classe che implementa l'interfaccia Iterator. Come Collection e Map, anche l'interfaccia Iterator e' in java.util. Un iteratore itera in una sola direzione (quindi non si puo' tornare indietro)²⁷, ed e' "usa e getta" (una volta arrivati alla fine non e' piu' utilizzabile). Ecco l'interfaccia Iterator:

²⁷In realta' ci sono alcune collezioni che hanno iteratori che sono in grado di tornare indietro, es. `listIterator()`

```

public interface Iterator<E> {
    boolean hasNext(); // restituisce true se c'e' un altro elemento nella collezione
    E next(); // ritorna l'elemento in posizione corrente e avanza, portando l'iteratore
               // nella posizione successiva
    void remove(); // Optional
}

public interface ListIterator<E> extends Iterator<E> { // rispetto a iterator classico ci
    permette anche di andare indietro

    boolean hasPrevious();

    E previous(); // proprio perche' ci permette di andare indietro ha questo metodo
                  previous(), oltre al next() che eredita da Iterator

    int nextIndex(); // restituisce il prossimo indice

    int previousIndex(); // restituisce l'indice precedente

    void set(E e); // Optional. Imposta il valore dell'elemento nella posizione corrente,
                   // cioe' dove ci troviamo con l'iteratore, al valore e.
}

```

Il programmatore e' in grado di utilizzare un iteratore relativo a qualsiasi collazione senza dover conoscere i dettagli implementativi della collezione.

Nota Importante: la dimensione della collezione su cui si itera NON deve essere modificata durante la scansione, altrimenti next() lancia un'eccezione (**unchecked²⁸VERIFICA**) per questo motivo. Idem previous() ovviamente. Eppure noi abbiamo un metodo remove() definito in Iterator... e infatti puoi usare remove() una sola volta quando scandisci con un iteratore: tu scandisci, poi usi remove e poi devi fermarti, perche' se fai next() questa ti lancia un'eccezione, dato che la collezione e' stata modificata. **MA QUINDI next() TI LANCIA L'ECCEZIONE A PRESCINDERE DA DOVE TU SIA DOPO AVER MODIFICATO LA COLLEZIONE E FATTO next()?** PERCHE' MAGARI E' COME IN C DOVE TU PUOI MODIFICARLA LA COLLEZIONE MENTRE TI MUOVI, PERO' POI DEVI RICALIBRARE LE DIMENSIONI DELLO SCANSIONAMENTO; OPPURE E' PROPRIO CHE QUAISIASI COSA TOCCHI, OVUNQUE TU SIA, TI LANCIA UN'ECCEZIONE?

Iteratori per Insiemi: gli iteratori per insiemi scandiscono i valori in un ordine che non si puo' prevedere, proprio perche' gli insiemi non sono ordinati.

Per ottenere un iteratore di una collezione possiamo invocare il metodo iterator() su un'istanza di Collection. Il metodo iterator() appartiene all'interfaccia Collection. Facciamo un'esempio di utilizzo degli iteratori con Collection:

```

Collection<E> c = ... // collezione di oggetti di tipo E
...
Iterator<E> it = c.iterator(); // iteratore per la collezione c
while (it.hasNext()) { // finche' il cursore non e' all'ultimo elemento
    E e = it.next(); // poni l'elemento corrente in e ed avanza
    ... // processa l'elemento corrente (denotato da e)
}

```

Questo e' il pattern che si utilizza quasi sempre. Vediamo un altro esempio, con una lista di elementi di tipo E:

```

List<E> c = ... // lista di oggetti di tipo E
...

```

²⁸Non controllata perche' abbiamo modo di controlare questa eccezione a tempo di scrittura del codice grazie alla buona programmazione: basta usare hasNext() e hasPrevious().

```

ListIterator<E> it = c.listIterator(c.size()); // iteratore per la lista c inizializzato
    per puntare all'ultimo elemento della lista

while (it.hasPrevious()) { //finche' il cursore non e' al primo elemento
    E e = it.previous(); // poni l'elemento corrente in e e sposta il cursore indietro
    ... // processa l'elemento corrente (denotato da e)
}

```

Vedremo che questi due cicli while con l'iteratore sono equivalenti a due cicli foreach.

26.1.3 Interfaccia Iterable:

Se una classe implementa Iterable allora e' iterabile. Una classe che implementa Iterable deve definire il metodo iterator() **CIOE' NON BASTA CHE LO EREDITI DEVE PROPRIO RIDEFINIRLO?** **CIOE' DEVE FARE OVERRIDING?**, astratto nell'interfaccia Iterable. Tutte le collezioni del JCF implementano Iterable. Ecco il codice di Iterable:

```

public interface Iterable<E> {
    Iterator<E> iterator();
}

```

Questo spiega perche' Collection abbia il metodo iterator(). La JCF estende automaticamente Iterable, quindi non dobbiamo scrivere noi `extends Iterable` a mano. Su tutte le classi che implementano Iterable e' possibile utilizzare il ciclo *foreach*, quindi anche su tutte le collezioni del JCF (ma ovviamente anche su altro, magari su roba che non e' neanche una collezione!). Basta che ti fai la tua classe e la dichiari figlia di Iterable e la ci puoi fare iterator(). Ecco un'esempio con la collezione ArrayList, classe che implementa l'interfaccia List, figlia di Collection:

```

List<Integer> l = new ArrayList<Integer>(new int[]{1, 2, 3}); //ArrayList implementazione
    di Lista. Vediamo che al costruttore di ArrayList passiamo un array di interi con
    questa sintassi: e' uno dei metodi per passare un array, puo' risultare utile
    for (Integer i : l) { //ciclo foreach: per tutti gli interi nella lista l fai questo
        System.out.println(i);
}

```

Notiamo che il ciclo foreach utilizzato qui sostituisce in pieno il ciclo while con l'iteratore utilizzato nei due esempi della sezione precedente. In realta' sotto il foreach chiama l'iterator, quindi l'utente non lo usa ma Java lo usa lo stesso.

26.2 Mappe:

Una mappa e' una struttura dati che memorizza le coppie [chiave, valore]²⁹. La chiave e' univoca all'interno di una mappa, il valore no. Vediamo il codice dell'interfaccia map:

```

public interface Map<K,V> { //prende due generics, uno per la chiave e uno per il valore

    int size();

    boolean isEmpty();

    void clear(); // Optional

    boolean equals(Object o); //confronto tra due mappe viene fatto confrontando le
        coppie chiave valore (coppia deve essere uguale a coppia e valore deve essere
        uguale a valore), quindi e' un equals() profondo

    boolean containsKey(Object key);

    boolean containsValue(Object value);

```

²⁹In python e C la chiamavamo dizionario.

```

V get(Object key);

V put(K key, V value); // Optional (l'istanza potrebbe arrivare dall'esterno, che so-
    da una base dati, quindi in quel caso non ha senso e magari vuoi proprio impedire
    che ci siano dei modi per modificare questa istanza, insomma tutto cio' che
    abbiamo gia' detto). Quindi non possiamo come in Python usare le parentesi quadre
    per inserire dei valori nella mappa, dobbiamo usare put

V remove(Object key); // Optional

Set<K> keySet(); //restituisce l'insieme delle chiavi, essendo le chiavi univoche
    posso metterle in un insieme, dato che non ci sono ripetizioni

Collection<V> values(); //restituisco la collezione dei valori, essendo i valori non
    univoci, non posso metterli in un insieme ma per forza in una
}

```

Ricordiamo che questo e' solo il *contratto d'interfaccia*, poi una classe concreta dovrà implementare questa mappa e potrà istanziare, perché Map non può istanziare direttamente essendo un'interfaccia.

26.3 Collezioni Ordinate:

Possiamo implementare delle collezioni ordinate, e già le abbiamo viste, almeno il nome e nel grafico. Stiamo parlando di:

- SortedSet: insiemi ordinati (comunque non sono ammesse ripetizioni)
- SortedMap: dizionari ordinati rispetto alla chiave

Ma a che servirebbe avere un insieme o un dizionario ordinato? Serve perché permette maggiore efficienza negli algoritmi. Comunque, ok le collezioni ordinate, ma per farlo **ho bisogno di definire un concetto di ordinamento!** Come definirlo? O implemento in una classe l'interfaccia Comparable (definisce sulla classe l'ordinamento) o creo una classe Comparator (classe esterna usata per confrontare due oggetti). Vediamo questi due modi uno alla volta:

- Implementare l'interfaccia Comparable:

```

public interface Comparable<T> {
    int compareTo(T o);
}

```

L'interfaccia quindi mi fa confrontare me stesso con altre istanze dello stesso tipo di dato. 0 significa oggetti uguali; negativo significa this. viene prima di o; positivo se viene dopo. Le slide propongono 0, -1 e 1, che sono l'implementazione banale di compareTo(), noi possiamo implementarlo in modo più interessante, ad esempio se abbiamo delle stringhe possiamo restituire la distanza lessicografica tra le due stringhe, cioè la somma delle differenze di ogni codice ASCII relativo ai caratteri tra le due stringhe. Quindi morale: se vuoi usare la classe in una collezione ordinata la classe deve implementare Comparable e quindi eredita' compareTo(). Esempio: ho la mia classe Persona e voglio poter ordinare le sue istanze in una collezione, allora devo fare public class Persona implements Comparable, così che erediti il metodo compareTo().

- Implementare l'interfaccia Comparator. Con cui, invece di implementare un'interfaccia sulla classe che è oggetto del confronto posso implementare una classe che fa da comparatore. Il codice di comparator è:

```

public interface Comparator<T> {
    int compare(T o1, T o2)
}

```

Quindi è come una classe esterna che ha il metodo compare() che prende in input i due oggetti che voglio confrontare; in questo modo non ho bisogno di invocare il metodo su uno

dei due oggetti che voglio confrontare ma li do entrambi come input. Quando serve sta cosa? Ad esempio quando ho una classe su cui non ho potere, quindi non posso estendere Comparable, allora utilizzo l'interfaccia Comparator e gli oggetti li do' come input a compare(). Oppure quando voglio creare dei comparatori *custom*: ad esempio ho delle stringhe e voglio confrontare le stringhe ma secondo una politica che voglio poter decidere io. Ma dato che la classe String ha già un suo modo di essere confrontata, implemento una classe esterna Comparator che mi permette di scrivere un nuovo confronto custom made. Il metodo compare() restituisce 0 se gli oggetti sono uguali, un negativo se o1 è prima di o2, un positivo se o1 è dopo di o2, sempre basandosi sulla politica di confronto che decidiamo.

Ok torniamo a SortedSet e SortedMap. Le classi che implementano queste due interfacce sono tipicamente dotate di due costruttori:

- Un costruttore senza argomento: suppone che le classi che lo usano implementano Comparable, e quindi basano il loro ordinamento sull'ordinamento definito da Comparable. Facciamo un esempio:

```
class Persona implements Comparable<Persona> {
    ...
    int compareTo(Persona o) { ... } //ordinamento per cognome e nome
        (ovviamente dobbiamo scriverlo noi questo compareTo(), Java non sa che
        le Persone vanno ordinate in questo modo
}
```

Quindi prendiamo la nostra classe Persona e gli facciamo implementare Comparable. A questo punto possiamo istanziare una collezione ordinata, per esempio una SortedSet implementata con TreeSet:

```
SortedSet<Persona> s1 = new TreeSet<Persona>();
```

Ecco la nostra istanza s1: un insieme di persone ordinate per cognome e nome

- Un costruttore con un argomento di tipo Comparator: basa il suo ordinamento sull'ordinamento definito dalla funzione compare del Comparator. Facciamo un esempio:

```
class PersonaComparatorEta implements Comparator<Persona> {
    int compare(Persona p1, Persona P2) //ordinamento per eta (anche qui
        ovviamente dobbiamo scriverlo noi questo ordinamento, mica Java lo
        conosce)
}
```

Quindi in questo caso creiamo una classe PersonaComparatorEta e gli facciamo implementare Comparator. A questo punto possiamo istanziare una collezione ordinata, per esempio una SortedSet implementata con TreeSet:

```
SortedSet<Persona> s2 = new TreeSet<Persona>(new PersonaComparatorEta());
```

Ecco la nostra istanza s2: un insieme di persone ordinate per eta'

26.4 Strutture Dati per l'implementazioni delle interfacce:

Su quali strutture dati sono basate le implementazioni delle classi? Ecco una tabella esplicativa:
Alcune osservazioni:

- Collection non è presente tra le interfacce perché non ha nessuna classe che la implementa direttamente, ma ha solo interfacce derivate.
- Non c'è l'interfaccia Queue ma soltanto la sua interfaccia derivata Deque perché Deque è più potente di Queue.

Ok, adesso andiamo a parlare di queste categorie di strutture dati:

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Table 1: Sull'asse verticale abbiamo le interfacce che andiamo a implementare, mentre sull'asse orizzontale delle categorie di strutture dati con cui implementare.

- HashTable: presenta dei bucket a cui sono collegati degli array, ad ogni bucket, quindi ad ogni array e' collegato un codice hash. Questa caratterizzazione rende le classi che implementano interfacce con il meccanismo HashTable molto efficienti per la lettura, meno per la scrittura. Le classi HashSet e HashMap, implementazioni rispettivamente dell'interfaccia Set e di Map, utilizzano il meccanismo HashTable.
- Resizable Array: e' il caso dell'implementazione di Studente, che aveva un array di esami di lunghezza 30, quello che succede e' che abbiamo una var che ci dice a che punto siamo arrivati nell'array (numEsamiFatti). se faccio piu' di 30 esami ho una funzione che fa un array piu' grosso e ci copia i valori di quello vecchio (anche se noi sta cosa di fare il realloc dell'array nell'implementazione di Studente non l'avevamo messa). Si parla quindi di un *array dinamico*. Liste e Doppie Code (Deque) utilizzano questo meccanismo.
- BalancedTree: autoespliattivo, mettiamo gli insiemi e le mappe (dato che Set e Map utilizzano questo meccanismo) in un *albero di ricerca bilanciato*, comodo per la ricerca e l'ordinamento. Precisamente in realta', sono SortedSet e SortedMap a utilizzare la classe TreeMap.
- LinkedList: liste collegate. Meccanismo utilizzato da Liste e Doppie Code.
- Hash Table + Linked List: presenta dei bucket a cui sono collegati delle liste collegate, invece che degli array. Quindi unisce questi due meccanismi. Viene utilizzato, come Hash Table, da Set e Map, cioe' da insiemi e mappe.

26.5 Domande su Wildcard e modifica collezione durante scansione con iteratore:

Buona sera, avrei due domande:

- Se, dato un tipo generico $C_j T_i$, una Wildcard $C_j ?_i$ permette di catturare tutte le possibili istanziazioni di T , il controllo sulla correttezza del tipo viene fatto a tempo di esecuzione? Se sì, allora cosa distingue l'utilizzo di una Wildcard dall'utilizzo di Object come classe contenitore di ogni oggetto (adottando quindi la consuetudine pre-Java5)? Io ho pensato che la differenza stia nella impossibilità di avere dati promiscui utilizzando le Wildcards, ma vorrei una conferma.

RISPOSTA: La sua intuizione e' corretta. Richiedendo come tipo di ritorno $C_j Object_i$ (**MA QUINDI $C_j Object_i$ sarebbe uguale a non utilizzare i generics bensì Object come prima di Java 5?**) stiamo dicendo che accettiamo anche collezioni di elementi promiscui. Indicando invece il generics $C_j ?_i$ stiamo dicendo che vogliamo una collezione di elementi omogenei, anche se non siamo in grado quando compiliamo di indicare quale sara' il tipo di questi elementi omogenei e vogliamo che sia il client a specificarlo.

- Nelle slide relative al JCF, precisamente nella slide n.34, c'e' scritto - in relazione all'utilizzo di un iteratore - che "la collezione su cui si itera non deve essere modificata durante l'iterazione, altrimenti alla successiva chiamata del metodo next(), l'iteratore lancia un'eccezione".
 - Per modifica, si intende modifica strutturale (eliminazione o aggiunta di un nodo) o modifica di qualsiasi tipo (anche una semplice assegnazione in un campo dati di un nodo)?

- L’eccezione viene lanciata a prescindere da dove il cursore si trovi? In altre parole: se faccio remove() durante la scansione, Java mi impedisce sempre di usare next() dopo oppure controlla se il cursore si trova in un punto in cui, anche se un nodo e’ stato rimosso, l’iteratore puo’ andare avanti? Faccio questa domanda perche’, volendo fare un parallelismo col C, mi ricordo che puntatori e iteratori potevano sempre avanzare anche se modificavamo la collezione.

RISPOSTA: Si parla di modifiche strutturali, quelle del contenuto dei nodi non hanno problemi. L’eliminazione strutturale e’ una modifica solitamente sicura, mentre la modifica distruttiva e’ l’aggiunta. In particolare la remove fatta sull’iterator e’ sicura mentre quella fatta utilizzando il metodo remove della collection puo’ creare problemi in presenza di piu’ thread (vedremo dopo durante il corso). In tal senso, il seguente codice passa indenne il primo ciclo mentre non sopravvive al secondo ciclo generando una eccezione del tipo ConcurrentModificationException.

```

import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedList;

public class ProvaIteratore {
    public static void main(String[] args) {
        LinkedList<String> a = new LinkedList<String>();
        a.addAll(Arrays.asList(new String[]{"A", "B"}));
        Iterator it = a.iterator();
        while (it.hasNext()) {
            it.next();
            it.remove();
        }
        System.out.println(a);
        a.addAll(Arrays.asList(new String[]{"A", "B"}));
        it = a.iterator();
        while (it.hasNext()) {
            a.add("K");
            it.next();
        }
        System.out.println(a);
    }
}

```

Esistono anche delle collezioni che in realta’ sono sicure rispetto alle modifiche durante l’iterazione, sono quelle che hanno Concurrent nel nome (es. ConcurrentHashMap)

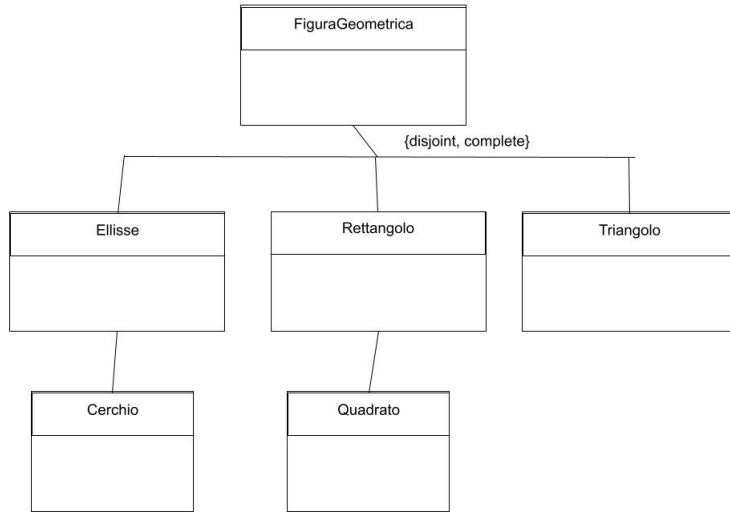
27 Figura Geometrica, Completa:

Realizzare le classi necessarie per modellare figure geometriche. Nel nostro dominio ci interessano le seguenti figure geometriche: rettangoli, quadrati, cerchi, ellissi, e triangoli. Definire un insieme di classi e una opportuna gerarchia partendo da una classe base FiguraGeometrica. Ogni figura geometrica e’ caratterizzata da una propria descrizione (una stringa) e da campi dati che consentono il calcolo dell’area. Ogni figura geometrica deve avere dei costruttori per inizializzare i valori delle proprieta’ e deve implementare le seguenti operazioni:

- double area() : restituisce l’area della figura geometrica
- double perimetro() : restituisce il perimetro della figura geometrica
- String toString() : per la rappresentazione come stringa delle informazioni della figura geometrica (descrizione e dati)

innanzitutto disegniamo in UML le classi:

Che, in Java, viene tradotto cosi’:



```

package Figure;

abstract public class FiguraGeometrica {

    private String descrizione;

    public FiguraGeometrica() {
        this.descrizione = "";
    }

    public FiguraGeometrica(String descrizione) {
        this.descrizione = descrizione;
    }

    public String getDescrizione() {
        return this.descrizione;
    }

    public void setDescrizione(String descrizione) {
        this.descrizione = descrizione;
    }

    abstract public double area();
    abstract public double perimetro();
    @Override
    abstract public String toString(); //cosi' impongo alle classi figlie di fare
                                     //overriding di toString()

}
  
```

```

package Figure;

public class Cerchio extends Ellisse {

    //eredita i due semiassi da ellisse: sono uguali e sono il raggio

    //eredita descrizione da FiguraGeometrica
    public Cerchio(double raggio) {
        super(raggio, raggio); //do due volte il raggio come due semiassi
    }
  
```

```

}

public Cerchio(double raggio, String descrizione) {
    super(raggio, raggio, descrizione); //do due volte il raggio come due semiassi
}

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

//eredita getDescrizione da FiguraGeometrica

//eredita perimetro e area da Ellisse

@Override
public String toString() {
    return "Descrizione: " + this.getDescrizione() + " " + "Raggio: " +
        this.getRaggio() + " " + "Perimetro: " + this.perimetro() + " " + "Area: "
        +
        this.area();
}

}

-----
package Figure;

public class Ellisse extends FiguraGeometrica {

    private final double semiassemaggiore;
    private final double semiasseminore;

    //descrizione la eredita da FiguraGeometrica

    public Ellisse(double semiassemaggiore, double semiasseminore) {

        this.semiassemaggiore = semiassemaggiore;
        this.semiasseminore = semiasseminore;
    }

    public Ellisse(double semiassemaggiore, double semiasseminore, String descrizione) {

        super(descrizione);

        this.semiassemaggiore = semiassemaggiore;
        this.semiasseminore = semiasseminore;
    }

    public double getSemiassemaggiore() {
        return this.semiassemaggiore;
    }

    public double getSemiasseminore() {
        return this.semiasseminore;
    }

    //get e set descrizione li eredita da FigureGeometriche
}

```

```

@Override
public double area() {

    return this.semiassemaggior * this.semiasseminore * Math.PI;
}

@Override
public double perimetro() {

    return (2 * Math.PI * Math.sqrt((semiassemaggior*semiassemaggior +
        semiasseminore*semiasseminore)/2));
}

@Override
public String toString() {
    return "Descrizione: " + this.getDescrizione() + " " + "Semiasse Maggiore: " +
        this.getsemiassemaggior() + " " + "Semiasse Minore: " +
        this.getsemiasseminore() +
        " " + "Perimetro: " + this.perimetro() + " " + "Area: " +
        this.area();
}

}

-----
package Figure;

public class Quadrato extends Rettangolo {

    //eredita descrizione da FiguraGeometrica e base e altezza da Rettangolo

    public Quadrato(double lato) {

        super(lato, lato); //do due volte il lato come base e come altezza
    }

    public Quadrato(double lato, String descrizione) {

        super(lato, lato, descrizione); //do due volte il lato come base e come altezza
    }

    public double getLato() {
        return this.getBase();
    }

    //eredita area, perimetro e toString da Rettangolo

    @Override
    public String toString() {
        return "Descrizione: " + this.getDescrizione() + " " + "Lato: " +
            this.getBase() + "Perimetro: " + this.perimetro() + " " + "Area: " +
            this.area();
    }
}

```

```
-----  
package Figure;  
  
public class Rettangolo extends FiguraGeometrica {  
  
    protected final double base;  
    protected final double altezza;  
  
    //descrizione la eredita da FiguraGeometrica  
  
    public Rettangolo(double base, double altezza) {  
  
        this(base, altezza, "");  
    }  
  
    public Rettangolo(double base, double altezza, String descrizione) {  
  
        super(descrizione);  
        this.base = base;  
        this.altezza = altezza;  
    }  
  
    public double getBase() {  
        return base;  
    }  
  
    public double getAltezza() {  
        return altezza;  
    }  
  
    //get e set descrizione li eredita da FiguraGeometrica  
  
    @Override  
    public double area() {  
        return base * altezza;  
    }  
  
    @Override  
    public double perimetro() {  
        return 2*base + 2*altezza;  
    }  
  
    @Override  
    public String toString() {  
        return "Descrizione: " + this.getDescrizione() + " " + "Base: " +  
            this.getBase() + " " + "Altezza: " + this.getAltezza() +  
            " " + "Perimetro: " + this.perimetro() + " " + "Area: " +  
            this.area();  
    }  
}  
-----  
package Figure;
```

```

public class Triangolo extends FiguraGeometrica {

    private final double lato1;
    private final double lato2;
    private final double lato3;

    //descrizione la eredita da FiguraGeometrica

    public Triangolo(double lato1, double lato2, double lato3) {

        this.lato1 = lato1;
        this.lato2 = lato2;
        this.lato3 = lato3;

    }

    public Triangolo(double lato1, double lato2, double lato3, String descrizione) {

        super(descrizione);

        this.lato1 = lato1;
        this.lato2 = lato2;
        this.lato3 = lato3;

    }

    public double getLato1() {
        return lato1;
    }

    public double getLato2() {
        return lato2;
    }

    public double getLato3() {
        return lato3;
    }

    //get e set descrizione le eredita da figura geometrica

    @Override
    public double area() { //formula di Erone

        double semiperimetro = (lato1 + lato2 + lato3)/2;

        double area = Math.sqrt(semiperimetro*(semiperimetro - lato1)*(semiperimetro - lato2)*(semiperimetro - lato3));

        return area;
    }

    @Override
    public double perimetro() {
        return lato1 + lato2 + lato3;
    }

    @Override
    public String toString() {
        return "Descrizione: " + this.getDescrizione() + " " + "Lato1: " + this.lato1 + "
           " + "Lato2: "
    }
}

```

```

        + this.lato2 + " " + "Lato3: " + this.lato3;
    }

}

-----
package Figure;

public class Main {

    public static double sommaAree(FiguraGeometrica[] f) {

        double sumArea = 0.d;

        for (int i = 0; i < f.length; i++) {
            sumArea += f[i].area(); //polimorfismo: il late binding permette a tempo di
                                   esecuzione di capire quale e' l'area giusta da usare
        }

        return sumArea;
    }

    public static void main(String[] args) {

        FiguraGeometrica[] f = new FiguraGeometrica[5];

        f[0] = new Cerchio(5, "non so");
        f[1] = new Ellisse(5, 4, "bho");
        f[2] = new Quadrato(3, "aio");
        f[3] = new Rettangolo(2, 10, "ciao");
        f[4] = new Triangolo(2, 4, 5, "hello");

        System.out.println(f[0]);
        System.out.println(f[1]);
        System.out.println(f[2]);
        System.out.println(f[3]);
        System.out.println(f[4]);

        System.out.print("L'area totale di queste figure e': ");
        System.out.println(sommaAree(f));
    }
}

```

Nota: `toString()` astratto: Notare come `FiguraGeometrica`, grazie al dichiarare `toString()` astratto, impone alle sue classi derivate di fare overriding di `toString()`. Questo vale in generale: vuoi imporre a le tue classi derivate di fare overriding di un certo metodo? Dichiaralo abstract nella classe madre.

Nota: stab e ooperazioni opzionali: Se un'operazione di una classe astratta e' opzionale per la classe concreta (non nel nostro caso), allora, quando l'IDE ci chiedera' di implementare noi facciamo un *metodo stab*, cioe' la implementiamo solo con un'eccezione, esempio:

```

public double area() {
    throw new UnsupportedOperationException(".....")
}

```

28 Esercizio Banca:

Realizzare le classi necessarie per rappresentare il seguente dominio applicativo. Una Banca e' un insieme ordinato di oggetti ContoCorrente. Di una banca interessa conoscere il nome e l'indirizzo (entrambe stringhe). Un conto corrente (i cui dettagli sono riportati in seguito) deve essere necessariamente di uno di due tipi : di debito o di credito. L'applicazione deve permettere di gestire la banca come insieme ordinato di conti, la contabilita' dei vari conti, e di stampare degli opportuni riepiloghi della situazione finanziaria dei conti e della banca. Un conto corrente e' caratterizzato da un codice (stringa, unico per il conto), dal saldo (intero), dal nome e cognome del proprietario (entrambi stringhe). Inoltre un conto corrente offre due operazioni per fare il deposito ed il prelievo dal conto:

- public void deposito(int cifra)
- public void prelievo (int cifra)

Un conto corrente viene creato inizializzando il suo codice e il saldo iniziale. Il nome e cognome del proprietario possono invece essere modificati successivamente ed in piu' occasioni. Un ContoCredito e' un particolare tipo di ContoCorrente, in cui viene pagata una commissione ad ogni operazione se il numero di operazioni effettuate supera una soglia prefissata:

- la soglia (un intero) puo' essere rappresentata come una costante
- il costo della commissione (da sottrarre dal saldo ad ogni operazione oltre soglia) e' una caratteristica del conto, che deve poter essere letta/modificata in ogni momento. Alla creazione di un conto si assume essere 0
- Deve essere disponibile un'operazione per azzerare il numero delle operazioni effettuate
- void reset()

Un ContoDebito e' un particolare tipo di ContoCorrente, in cui vengono riconosciuti degli interessi (che si sommano al saldo attuale) quando ritenuto opportuno:

- void riconosciInteresse(double interesse)

prende un valore di interesse – nel range [0..1] e lo applica al saldo attuale, aumentando il saldo della cifra così' ottenuta (con arrotondamento) Ad es., se il saldo attuale e' 50, riconosciInteresse(0.65) fa sì' che il nuovo saldo sia 82. Si vuole poter confrontare due conti correnti in base al loro saldo, cosa necessaria anche per il mantenimento dell'ordinamento dei conti correnti nella banca. Creare l'applicazione con il metodo main in modo che crei almeno 3 conti (di tipi differenti) e dopo averli operati, stampi la situazione finanziaria della banca. Verificare come la banca rimane ordinata, facendo varie prove che cambiano l'ordine di inserimento dei conti nella banca e l'ordine con cui l'operatività del conto si succede all'inserimento nella banca.

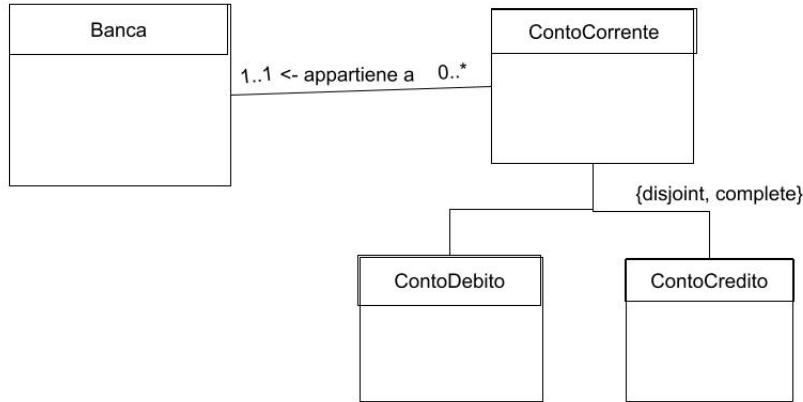
Risposta: innanzitutto il grafico UML: Dopodiché andiamo a scrivere il codice:

```
package sistemabancario;

import java.util.Arrays;

public class Banca { //vedremo poi come modificare questo esercizio con la JCF
    private final String nome;
    private String indirizzo;
    private ContoCorrente[] conti;
    private int index; //ci permette di tenere conto di dove siamo nell'array
    private static final int MAX = 10; //ci dice la grandezza del nostro array iniziale

    public Banca(String nome, String indirizzo) {
```



```

        this.nome = nome;
        this.indirizzo = indirizzo;
        this.conti = new ContoCorrente[MAX];
        this.index = 0;
    }

    public void aggiungiConto(ContoCorrente c) {

        if (index == MAX) {
            throw new RuntimeException("Numero massimo di conti raggiunto per questa
                banca"); //lanciamo una RuntimeException perche' e' unchecked e non
                dobbiamo gestirla
        }

        this.conti[index] = c;
        index++;
        //aggiunto questo nuovo elemento all'array, adesso dobbiamo ordinare quest'ultimo
        this.ordinaConti();
    }

    public void ordinaConti() {
        ContoCorrente[] buffer = Arrays.copyOf(this.conti, index); //copiamo l'array conti
        fino alla posizione index
        Arrays.sort(buffer); //buffer ora e' un array ordinato, ordinato rispetto al
        criterio
        //che gli daremo definendo compareTo() nella classe ContoCorrente

        for (int i = 0; i < index; i++) {
            this.conti[i] = buffer[i];
        }
    }
}

```

```

public int getBilancioTotale() {

    int somma = 0;
    for(ContoCorrente conto: conti) {
        if (conto != null) {
            somma += conto.getSaldo();
        }
    }
    return somma;
}

public String getNome() {
    return nome;
}

public String getIndirizzo() {
    return indirizzo;
}

@Override
public String toString() {

    String buffer = "Nome Banca: " + this.getNome() + " " +
                   "Indirizzo: " + this.getIndirizzo() + " " + "Conti: ";

    for (int i = 0; i < index; i++) {
        buffer += conti[i] + " ";
    }

    return buffer;
}

}

-----
package sistemabancario;

/**
 *
 * @author biar
 */
abstract public class ContoCorrente implements Comparable<ContoCorrente> {

    private final String codice;
    protected int saldo;
    private String nome;
    private String cognome;

    public ContoCorrente(String codice, int saldoIniziale) {
        this.codice = codice;
        this.saldo = saldoIniziale;
    }

    public String getCodice() {
        return this.codice;
    }
}

```

```

public int getSaldo() {
    return this.saldo;
}

public String getNome() {
    return this.nome;
}

public String getCognome() {
    return this.cognome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public void setCognome(String cognome) {
    this.cognome = cognome;
}

public abstract void deposito(int cifra); //SI DOVREBBE PROVARE A IMPLEMENTARE ->
public abstract void prelievo(int cifra); // -> ordinaConti() anche per deposito e
    prelievo

@Override
public abstract String toString();

@Override
public int compareTo(ContoCorrente c) {

    if (this.saldo > c.saldo) {
        return 1;
    } else if (this.saldo < c.saldo) {
        return -1;
    } else {
        return 0;
    }

}

}

-----
package sistemabancario;

/**
 *
 * @author biar
 */
public class ContoCredito extends ContoCorrente {

    //eredita: codice, saldo, nome e cognome
    private static final int SOGLIA = 3;
    private int commissione;
    private int numOperazioni;

    public ContoCredito(String codice, int saldo) {

        super(codice, saldo);

        commissione = 0;
        numOperazioni = 0;
    }
}

```

```

        public int getCommissione() {
            return commissione;
        }

        public int getSOGLIA() {
            return SOGLIA;
        }

        public void setCommissione(int commissione) {
            this.commissione = commissione;
        }

        public void reset() {
            this.numOperazioni = 0;
        }

    @Override
    public void deposito(int cifra) {

        if (numOperazioni > SOGLIA) {
            this.saldo -= commissione;
        }

        this.saldo += cifra;

        numOperazioni++;

    }

    @Override
    public void prelievo(int cifra) {

        if (numOperazioni > SOGLIA) {
            this.saldo -= commissione;
        }

        this.saldo -= cifra;

        numOperazioni++;

    }

    @Override
    public String toString() {
        return "ContoCredito: " + this.getNome() + " " + this.getCognome()
            + " " + "Saldo:" + this.getSaldo() + " " + "Commissione: " +
            this.getCommissione() +
            " " + "Soglia:" + this.getSOGLIA();
    }
}

}

-----
package sistemabancario;

/**
 *
 * @author biar
 */
public class ContoDebito extends ContoCorrente {

```

```

public ContoDebito(String codice, int saldo) {
    super(codice, saldo);
}

@Override
public void deposito(int cifra) {
    saldo += cifra;
}

@Override
public void prelievo(int cifra) {
    saldo -= cifra;
}

public void riconosciInteresse(double interesse) {
    saldo *= 1+interesse;
}

@Override
public String toString() {
    return "ContoDebito: " + this.getNome() + " " + this.getCognome()
           + " " + "Saldo:" + this.getSaldo() + " ";
}
}

-----
package sistemabancario;

public class Main {

    public static void main(String[] args) {

        ContoCredito primo = new ContoCredito("GCMHERB", 200);
        primo.setNome("Giovanni");
        primo.setCognome("Ghilberti");
        primo.setCommissione(3);

        ContoDebito secondo = new ContoDebito("FUYJERT", 150);
        secondo.setNome("Sandro");
        secondo.setCognome("Sandri");

        ContoDebito terzo = new ContoDebito("QAMGBNB", 700);
        terzo.setNome("Jesus");
        terzo.setCognome("Rodriguez");

        Banca banca1 = new Banca("Monte dei Saschi di Piena", "indirizzo1");

        banca1.aggiungiConto(primo);

        System.out.println(banca1);

        banca1.aggiungiConto(secondo);

        System.out.println(banca1);

        banca1.aggiungiConto(terzo);

        System.out.println(banca1);
    }
}

```

```

terzo.prelievo(400);
primo.deposito(500);
secondo.deposito(50);

ContoDebito quarto = new ContoDebito("VLKHABVI", 875);

banca1.aggiungiConto(quarto);

quarto.setNome("Ghiaf");
quarto.setCognome("Azele");

System.out.println(banca1);

secondo.deposito(999999999);

ContoDebito quinto = new ContoDebito("VLKHABVI", 875);

banca1.aggiungiConto(quinto);

quinto.setNome("Gigi");
quinto.setCognome("Pop Smoke");

System.out.println(banca1);

}

}

```

ATTENZIONE IMPORTANISSIMO: l'ordinamento viene effettuato soltanto al momento dell'inserimento del conto nella banca giusto? Eh ma quindi non viene mantenuto questo ordinamento se io modifco il saldo dei conti già presenti nell'array! *Questo vale in generale per qualsiasi collection dentro il JCF, se abbiamo degli oggetti ordinati in base a un parametro, non e' che, una volta inseriti, modificando il parametro l'oggetto si sposta nella collezione per riordinarsi!* Se vogliamo riordinare la collezione in questo caso dobbiamo fare delle **attivita' complesse**: nel nostro caso l'attivita' complessa sarebbe imporre che l'array si riordini dopo ogni prelievo/deposito. Vedremo piu' avanti nel corso le attivita'. Una maniera piu' "sporca" per risolvere questo problema senza usare le attivita' complesse potrebbe essere quella di dare al ContoCorrente da cui si preleva/deposita un riferimento alla Banca a cui il ContoCorrente appartiene. Così che sia possibile scrivere nei metodi prelievo() e deposito() del ContoCorrente il metodo ordinaConto() appartenente alla Banca: in questo modo riusciamo a implementare l'ordinamento dei conti ogni volta che c'e' un prelievo o un deposito, *ma e' una procedura piu' sporca che non andrebbe fatta, perche' riduce l'incapsulamento.*

28.1 Modifica esercizio Banca con JCF:

Modificare l'implementazione dell'esercizio Sistema Bancario usando una lista (interfaccia List del JCF) per rappresentare la collezione dei conti correnti della classe Banca:

```

public class Banca {

    private final String nome;
    private final String indirizzo;

    //versione con la lista:
    private List<ContoCorrente> conti; //List e' un'interfaccia! Non puo' essere
        istanziata: dovremo usare un'implementazione di List, ma lasciamo la variabile a
        List cosi' che se cambiamo implementazione comunque questo non va cambiato.

```

```

public Banca (String nome, String indirizzo) {

    this.nome = nome;
    this.indirizzo = indirizzo;

    conti = new LinkedList<ContoCorrente>(); // vediamo che al posto di istanziare
    l'array istanziamo una Lista, precisamente usiamo l'implementazione
    LinkedList di List

}

public void aggiungiConto(ContoCorrente c) {

    //la lista diversamente dall'array di prima non ha una grandezza massima, quindi non
    dobbiamo controllare il numero massimo di conti possibili

    conti.add(c);
    java.util.Collections.sort(conti); //metodo utilizzabile solo con le liste

}

//tutto il resto non va modificato: gli altri metodi sono uguali a quelli del caso
dei conti con l'array

}

```

Esercizio Banca con TreeSet, ma TreeSet fa schifo! Modificare la classe Banca usando degli insiemi (interfaccia Set del JCF) per implementare la collezione dei conti correnti:

```

public class Banca {

    private final String nome;
    private final String indirizzo;

    //versione con l'insieme:
    private Set<ContoCorrente> conti; //Set e' un'interfaccia! Non puo' essere
    istanziata: dovremo usare un'implementazione di Set, ma lasciamo la variabile a
    Set cosi' che se cambiamo implementazione comunque questo non va cambiato.

    public Banca (String nome, String indirizzo) {

        this.nome = nome;
        this.indirizzo = indirizzo;

        conti = new TreeSet<ContoCorrente>(); // usiamo l'implementazione TreeSet di Set,
        si tratta dell'implementazione ORDINATA del Set. Cio' vuol dire che qui
        l'ordinamento NON deve essere fatto in maniera esplicita come invece abbiamo
        fatto per la lista (difatti non esiste neanche una funzione sort per i Set).
        Tra l'altro non esistono implementazioni ordinate di List, le uniche
        implementazioni ordinate che il JCF offre sono del Set (TreeSet appunto) e
        del Map (TreeMap)

    }

    public void aggiungiConto(ContoCorrente c) {

        conti.add(c); //siccome il TreeSet e' gia' ordinato mi basta solo l'add per avere
        tutto e bello ordinato (ovviamente ordinato quando aggiungo i valori, non
        quando li modifco!!!)
    }
}

```

}

}

Il discorso e' che l'unica cosa che - essendo questo insieme ordinato - l'unica cosa che lo differenzia da un'implementazione con Lista sarebbe il fatto di non poter aggiungere elementi gia' presenti nell'insieme. **Ma c'e' un problema**, il codice di TreeSet del JCF e' noto alla community per essere stato scritto male: oggetti che non dovrebbero essere aggiunti al TreeSet perche' gia' presenti vengono aggiunti lo stesso. Andando a studiare il codice del TreeSet, si puo' infatti vedere che esso fa il controllo dei saldi soltanto su base compareTo() e **supponendo che la struttura dati sia gia' in uno stato ordinato**. Questo crea enormi problemi. Facciamo un esempio: ho 3 conti correnti, con i seguenti 3 saldi, ordinati:

13 15 17

aggiorno il secondo conto, levando 10. Sappiamo che non dobbiamo aspettarci che il conto 15 - 10 = 5 venga ordinato in prima posizione perche' le strutture del JCF sono ignare di queste modifiche, e ordinano solo quando si aggiunge un elemento. Il TreeSet non e' piu' ordinato, quindi resta:

13 5 17

e fino a qui tutto ok, gia' lo sapevamo. Qua arriva il nuovo problema: provando a reinserire il secondo conto (quello con 5 di saldo), gia' presente nel TreeSet, ci aspetteremmo che esso non venga riaggiunto e che il TreeSet venga ordinato, giusto? E invece no, e la situazione diventa questa:

5 13 5 17

Perche'? Alla fine il TreeSet abbiamo detto che fa il controllo con il compareTo(), che restituisce 0 se il saldo e' lo stesso, e quindi dovrebbe accorgersi che ci sono due saldi uguali! Il problema non e' infatti nel compareTo(), ma nel fatto che il TreeSet **suppone che la struttura dati sia gia' in uno stato ordinato**. Cio' significa che quando vado a inserire il secondo conto, lui il primo elemento che vede e' il conto che ha 13 di saldo, e poiche' $13 > 5$, poiche' pensa che l'insieme sia gia' ordinato, e quindi che dopo il conto con saldo 13 ci siano solo conti con saldo maggiore di 13, assume che nessun conto con saldo 5 sia presente, e quindi aggiunge il conto con 5 all'inizio. Quindi il TreeSet e' una mezza monnezza. L'implementazione del Set con L'Hashset e' meglio perche' il confronto non lo fa col compareTo() ma sull>equals e l=hashcode, quindi non c'e' un ordinamento che possa supporre e deve per forza di cose percorrere tutta la struttura per verificare se l'elemento e' gia' presente.

Modificare l'astrazione entita' Studente utilizzando un'insieme (interfaccia Set del JCF) per implementare la collezione di esami. **CE LO LASCIA DA FARE A NOI**.

29 Laboratorio 6: Implementazione SS di MioSet e MiaLista

Questo e' MioSet, col suo iteratore:

```
package laboratorio6;

import java.lang.reflect.Array;
import java.util.Collection;
import java.util.Iterator;
import java.util.Objects;
import java.util.Set;

class NodoSet<E> {
```

```

private E info;
NodoSet<E> next;

public NodoSet (E info, NodoSet<E> next) {
    this.info = info;
    this.next = next;
}

public NodoSet<E> next() {
    return this.next;
}

public void setNext(NodoSet<E> next) {
    this.next = next;
}

public E getInfo() {
    return info;
}

}

public class MioSet<E> implements Set<E>, Cloneable {

    private NodoSet<E> inizio;

    @Override
    public int size() {

        int retValue = 0;

        NodoSet<E> nodoCorrente = this.inizio;

        while (nodoCorrente != null) {

            retValue += 1;

            nodoCorrente = nodoCorrente.next();
        }

        return retValue;
    }

    @Override
    public boolean isEmpty() {

        return this.inizio == null;
    }

    @Override
    public boolean contains(Object o) {

        if (this.inizio != null && o.getClass() == this.inizio.getInfo().getClass()) {

            NodoSet<E> nodoCorrente = inizio;

            while (nodoCorrente != null) {

                if (nodoCorrente.getInfo().equals(o)) {

```

```

        return true;
    }

    nodoCorrente = nodoCorrente.next();
}

return false;
}

@Override
public Iterator<E> iterator() {
    return new MioSetIterator<>(this.inizio);
}

@Override
public Object[] toArray() {

    Object[] retValue = new Object[this.size()];

    int index = 0;

    for (E e: this) {

        retValue[index] = e; //poteva essere scritto come retValue[index++] = e;

        index++;
    }

    return retValue;
}

@Override
public <T> T[] toArray(T[] a) { //e' un overloading (anche se cambia il tipo di
    ritorno non e' importante),
                           //devo per forza mettergli in input un parametro
                           //perche' due metodi overloaded non possono
                           //variare solo per il tipo di ritorno ma devono
                           //avere numero o tipo dei parametri diverso

    T[] retValue = (T[]) Array.newInstance(a.getClass().getComponentType(),
                                             this.size());
    //NON SI POSSONO CREARE ARRAY DI TIPO GENERICO, QUINDI USIAMO QUESTO METODO

    //la classe Array e' nel package reflect, che e' subpackage di java.lang
    //ma come sappiamo un subpackage non implica che importando il superpackage venga
    //importato anche lui
    //quindi non possiamo non importare nulla (come avremmo fatto con java.lang), ma
    //dobbiamo importare java.lang.reflect

    int index = 0;

    for (E e: this) {

        retValue[index] = (T)e;

        index++;
    }

    return retValue;
}

```

```

@Override
public boolean add(E e) {

    if (!this.contains(e)) {

        NodoSet<E> nuovoNodo = new NodoSet<>(e, this.inizio);

        nuovoNodo.setNext(this.inizio);

        this.inizio = nuovoNodo;

        return true;
    }

    return false;
}

@Override
public boolean remove(Object o) {

    if (!this.contains((E)o)) {
        return false;
    }

    else {

        if (inizio.getInfo().equals(o)) {
            inizio = inizio.next();

            return true;
        }

        for (NodoSet <E> nodoCorrente = inizio; nodoCorrente.next() != null;
            nodoCorrente = nodoCorrente.next()) {

            if (nodoCorrente.next().getInfo().equals(o)) {
                nodoCorrente.setNext(nodoCorrente.next().next());

                return true;
            }
        }
    }

    return false;
}

@Override
public void clear() {
    this.inizio = null; //il Garbage Collector dealloca la memoria
}

```



```

@Override
public boolean equals(Object o) { //e' la stessa implementazione di equals di
    InsiemeSS

    if (o == null || !this.getClass().equals(o.getClass())) {
        return false;
    }

    MioSet<E> oCasted = (MioSet<E>)o;

```

```

//controlliamo che tutti gli elementi di oCasted siano in this

for (NodoSet<E> nodoCorrente = oCasted.inizio; nodoCorrente != null; nodoCorrente =
= nodoCorrente.next()) {

    if (!this.contains(nodoCorrente.getInfo())) {
        return false;
    }
}

//controlliamo che tutti gli elementi di this siano in oCasted

for (NodoSet<E> nodoCorrente = this.inizio; nodoCorrente != null; nodoCorrente =
nodoCorrente.next()) {

    if (!oCasted.contains(nodoCorrente.getInfo())) {
        return false;
    }
}

//se ha superato entrambi i controlli allora sono uguali
return true;
}

@Override
public int hashCode() {
    int hash = 0;

    for (E e: this) { //questo for each sotto chiama l'iteratore
        hash += hash + Objects.hashCode(e);
    }

    return hash;
}

@Override
public Object clone() {

    try {

        MioSet<E> copia = (MioSet<E>) super.clone(); //copia superficiale

        if (this.size() > 0) {

            NodoSet<E> primoNodo = new NodoSet<>(this.inizio.getInfo(), null);
            NodoSet<E> ultimoNodo = primoNodo;

            for (NodoSet<E> nodoCorrente = this.inizio.next(); nodoCorrente != null;
                nodoCorrente = nodoCorrente.next()) { //iniziamo da this.inizio.next()
                perche' il primo nodo gia' lo abbiamo messo

                ultimoNodo.next = new NodoSet<>(nodoCorrente.getInfo(), null);

                ultimoNodo = ultimoNodo.next();
            }

            copia.inizio = primoNodo;
        }
    }
}

```

```

        return copia;

    } catch(CloneNotSupportedException e) {
        throw new InternalError(e.toString());
    }

}

@Override
public String toString() {

    String buffer = "(";

    for (E e: this) { //questo for each chiama l'iteratore sull'istanza this e
        restituisce l'elemento e
        buffer += " " + e;
    }

    return buffer + ")";
}

//Bulk Operation

@Override
public boolean containsAll(Collection<?> c) { //collezione di Object

    for (Object o: c) { //itera sugli elementi di c, uso Object perche' non so ? che
        tipo sia
            //quindi nel dubbio metto Object che tanto lo sara' sempre e
            per forza
            //DOMANDA: c'e' un altro modo di farlo, senza il for each?

        if (!this.contains(o)) {
            return false;
        }
    }

    return true;
}

@Override
public boolean addAll(Collection<? extends E> c) {

    boolean retValue = false;

    for (E e: c) { //gli elementi di c sono di tipo ? che estende la classe E, quindi
        sono anche di tipo E
            //per questo posso chiamarli col tipo E. La loro classe piu'
            specifica e' pero' ?.

        retValue = retValue || this.add(e);
    }

    return retValue;
}

@Override
public boolean retainAll(Collection<?> c) {

```

```

        boolean retValue = false;

        for (E e: this) { //vado sul sicuro dicendo che gli elementi di tipo ? sono Object

            if (!c.contains(e)) {
                retValue = retValue || this.remove(e);
            }
        }

        return retValue;
    }

    @Override
    public boolean removeAll(Collection<?> c) {

        boolean retValue = false;

        for (Object o: c) {

            retValue = retValue || this.remove(o); //ma questo chiama il remove di Object
            // o di MioSet???
        }

        return retValue;
    }

}

```

```

package laboratorio6;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class MioSetIterator<E> implements Iterator<E> {

    NodoSet<E> attuale;

    public MioSetIterator(NodoSet<E> attuale) {
        this.attuale = attuale;
    }

    public MioSetIterator() {
        this.attuale = null;
    }

    @Override
    public boolean hasNext() {
        return attuale != null; //uso la var next invece che next() cosi' da non
        //confondere il next()
    }                                //di MioSetIterator con il next() di NodoSet

    @Override
    public E next() {

```

```

        if (!this.hasNext()) {
            throw new NoSuchElementException();
        }

        E retValue = attuale.getInfo();

        this.attuale = this.attuale.next;

        return retValue;
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}

```

Invece questa e' MiaLista, con il suo iteratore:

```

package laboratorio6;

import java.lang.reflect.Array;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class MiaLista<E> implements List<E>, Cloneable { //con side-effect, senza
    condivisione di memoria

    private Object[] array;
    int size;

    public MiaLista() {

        size = 0;

        array = (Object[]) Array.newInstance(this.getClass().getComponentType(), 13);
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean add(E element) {

        if (size >= this.array.length) {

            array = (Object[]) Arrays.copyOf(this.array, this.array.length*2);
        }
    }
}

```

```

        array[size++] = element;

        return true;
    }

@Override
public void add(int index, E element) {

    if (index < 0 || index >= size ) {
        throw new IndexOutOfBoundsException();
    }

    if (size >= this.array.length) {

        array = (Object[]) Arrays.copyOf(this.array, this.array.length*2);

    }

    for (int i = size-1; i >= index; i--) {
        //trasliamo di uno tutti gli elementi da index in poi cosi' da creare lo spazio
        //per mettere l'elemento
        //partiamo da size-1 e scendiamo

        this.array[i+1] = this.array[i];
    }

    this.array[index] = element;
    size++; //traslando avanti per lasciare uno spazio abbiamo aumentato di 1 la
            //dimensione
}
}

@Override
public boolean remove(Object o) {

    if (!this.contains(o)) {
        return false;
    }

    remove(indexOf(o));

    return true;
}

@Override
public E remove(int index) {

    if (index < 0 || index >= size ) {
        throw new IndexOutOfBoundsException();
    }

    Object retVal = array[index];

    for (int i = index; i < size-1; i++) {

        this.array[i] = this.array[i+1];
    }

    size--;
}

if (size < this.array.length/2 && array.length > 13) {
}

```

```

        array = (Object[]) Arrays.copyOf(this.array, this.array.length/2);
    }

    return (E) retValue;
}

@Override
public void clear() {
    this.array = new Object[13];
    size = 0;
}

@Override
public E get(int index) {

    if (index < 0 || index >= size ) {
        throw new IndexOutOfBoundsException();
    }

    return (E) array[index];
}

@Override
public E set(int index, E element) {

    if (index < 0 || index >= size ) {
        throw new IndexOutOfBoundsException();
    }

    Object retValue = this.array[index];
    this.array[index] = element;

    return (E) retValue;
}

@Override
public int indexOf(Object o) {

    for (int i = 0; i < size; i++) {

        if (this.array[i].equals(o)) {
            return i;
        }
    }

    return -1;
}

@Override
public int lastIndexOf(Object o) {

    for (int i = size-1; i > -1; i--) {

        if (this.array[i].equals(o)) {
            return i;
        }
    }
}

```

```

        return -1;
    }

    @Override
    public List<E> subList(int fromIndex, int toIndex) { //fromIndex incluso toIndex
        escluso

        if (fromIndex < 0 || fromIndex >= size || toIndex < 0 || toIndex >= size ||
            fromIndex > toIndex) {
            throw new IndexOutOfBoundsException();
        }
    }

    MiaLista<E> subLista = new MiaLista<E>();

    for (int i = toIndex; i < fromIndex; i++) {

        subLista.add((E)this.array[i]);
    }

    return subLista;
}

@Override
public boolean isEmpty() {
    return size == 0;
}

@Override
public boolean contains(Object o) {

    for (Object elementoCorrente: this.array) { //LA PROF QUA HA FATTO IL CICLO FOR
        NORMALE
        if(elementoCorrente.equals(o)) {
            return true;
        }
    }

    return false;
}

@Override
public Iterator<E> iterator() {
    return new MiaListaIterator<>(this.array, this.size);
}

@Override
public String toString() {

    return Arrays.toString(Arrays.copyOf(this.array, this.size));
}

@Override
public boolean equals(Object o) {

    if(o == null || !this.getClass().equals(o.getClass())) {
        return false;
    }
}

```

```

}

MiaLista oCasted = (MiaLista) o;

Iterator<E> itThis = this.iterator();
Iterator<?> itOCasted = oCasted.iterator();

while (itOCasted.hasNext() && itThis.hasNext()) { //finche' tutti e due hanno elementi

    E elemThis = itThis.next();
    Object elemOCasted = itOCasted.next();

    if (elemThis == null && elemOCasted != null) { //facciamo questo controllo perche' dopo non possiamo
        return false; //invocare equals su null, mentre come parametro invece se
    } //possiamo passarlo, quindi se e' elemOCasted a essere null
        //se ne accorgera' equals subito dopo. Il problema e' solo //evitare che sia null l'oggetto su cui invochiamo equals
    if (elemThis != null && !elemThis.equals(elemOCasted)) {
        return false;
    }

    //if (!(o1 == null ? o2 == null : o1.equals(o2))) //Si poteva fare piu' //return false //compattamente cosi'
}

return !(itThis.hasNext() || itOCasted.hasNext()); //restituisce true se entrambe hanno finito gli elementi
}

@Override
public int hashCode() {
    return 1;
}

@Override
public Object clone() {

    try {

        MiaLista<E> copia = (MiaLista<E>) super.clone();

        copia.array = this.array.clone(); //chiama il clone di Object[]
        copia.size = this.size;

        return copia;
    } catch(CloneNotSupportedException e) {
        throw new InternalError(e.toString());
    }
}

@Override
public Object[] toArray() {
}

```

```

        throw new UnsupportedOperationException("Not supported yet."); //To change body of
        generated methods, choose Tools | Templates.
    }

    @Override
    public <T> T[] toArray(T[] a) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of
        generated methods, choose Tools | Templates.
    }

//Bulk Operation

    @Override
    public boolean containsAll(Collection<?> c) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of
        generated methods, choose Tools | Templates.
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of
        generated methods, choose Tools | Templates.
    }

    @Override
    public boolean addAll(int index, Collection<? extends E> c) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of
        generated methods, choose Tools | Templates.
    }

    @Override
    public boolean removeAll(Collection<?> c) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of
        generated methods, choose Tools | Templates.
    }

    @Override
    public boolean retainAll(Collection<?> c) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of
        generated methods, choose Tools | Templates.
    }

    @Override
    public ListIterator<E> listIterator() {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of
        generated methods, choose Tools | Templates.
    }

    @Override
    public ListIterator<E> listIterator(int index) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body of
        generated methods, choose Tools | Templates.
    }

}

```

```
-----  
package laboratorio6;  
  
import java.util.Iterator;  
import java.util.NoSuchElementException;  
  
public class MiaListaIterator<E> implements Iterator<E> {  
  
    private Object[] array;  
    private int size;  
    private int index;  
  
    public MiaListaIterator(Object[] array, int size) {  
  
        this.array = array;  
        this.size = size;  
        this.index = 0;  
    }  
  
    @Override  
    public boolean hasNext() {  
        return index < size;  
    }  
  
    @Override  
    public E next() {  
  
        if (!this.hasNext()) {  
            throw new NoSuchElementException();  
        }  
  
        Object retValue = this.array[index];  
  
        index++;  
  
        return (E) retValue;  
    }  
}
```

Part VII

Interfacce Grafiche:

30 JFC e Swing:

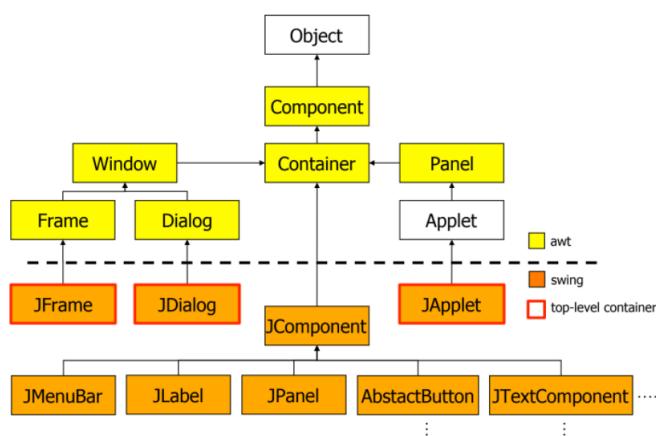
La JFC (Java Foundation Classes) e' un framework per lo sviluppo di GUI (Graphical User Interface, cioe' interfacce grafiche). Quindi come JFC ci da' una serie di classi per le collezioni JFC ci da' una serie di classi per lo sviluppo di interfacce grafiche (si intende interfacce grafiche desktop!!!). Le sue Componenti Swing sono quelle che permettono la realizzazione di interfacce grafiche. Oltre a queste la JFC permette di configurare l'aspetto grafico (supporto per diversi Look-and-Feel) delle GUI, supporta tecnologie assistive (es. Braille), permette di integrare immagini in 2D e di adattare l'input e l'output a piu' lingue (multilingua). Noi pero' in questo corso di concentriamo solo sulle Componenti Swing di JFC, che permettono la creazione di GUI.

30.1 Java Swing:

Toolkit di Java per le GUI. Indipendente dalla piattaforma sottostante (le componenti Swing sono implementate da Java e non usano il toolkit grafico del SO sottostante, quindi le GUI di Java Swing sono uguali in ogni SO), personalizzabile e basata sul pattern Model/View/Controller (MVC)³⁰ con paradigma di programmazione basato su eventi (event-driven) a singolo thread (single-threaded). V. dopo per capire quest'ultima parte. La libreria Swing e' parte della piattaforma Java Standard Edition (Java SE) dalla 1.2, ed e', *in realta', un'estensione di AWT (Abstract Window Toolkit)*.

- AWT: ha componenti heavyweight (pesanti). Sono pesanti perche' ogni componente viene gestito da un corrispondente componente nativo del Sistema Operativo sottostante. Quindi per questo e' chiaro che il comportamento dei componenti di AWT e' system-dependent e che alla creazione di un componente AWT corrisponde l'allocazione di risorse nel toolkit grafico del Sistema Operativo
- Swing: ha invece componenti lightweight (leggeri). Sono leggeri perche' implementati direttamente in Java, quindi non sono system-dependent: ad esempio il rendering grafico e' affidato a Java 2D, non al toolkit grafico del SO. Per questo, diversamente dall'AWT la creazione di un componente Swing non corrisponde all'allocazione di risorse nel toolkit grafico del SO

La libreria Swing include 18 package pubblici, i cui due principali sono `javax.swing.*` e `javax.swing.event.*`. I componenti Swing sono organizzati nella seguente gerarchia, che, come vediamo, estende quella di AWT (quella in giallo):



Partiamo dall'alto, sotto Object: l'interfaccia Component ha come figlia l'interfaccia Container. L'interfaccia Container specifica qualsiasi oggetto sullo schermo che puo' contenere altri oggetti. Gli oggetti che possono contenere altri oggetti, cioe' le specializzazioni di Container, possono essere:

- Window (finestra), una finestra puo' essere:

³⁰Si tratta di un design pattern separa in tre parti l'applicazione: il model si occupa della gestione dei dati (base dati), il view si occupa dell'interfaccia grafica (opposto del model), il controller si occupa di unire il view e il model

- Una Dialog: finestra di pop-up, disabilita la finestra genitrice; impongono quindi di lavorare su loro prima di continuare il lavoro. La classe Dialog ha come implementazione in Swing la Classe JDialog
- Un Frame: finestre che vivono per cavoli loro, non devi per forza lavorare prima su loro, ma puoi lavorare su piu' Frame contemporaneamente. La classe Frame ha come implementazione in Swing la Classe JFrame
- Panel

Le altre due implementazioni di Swing mancanti sono JApplet, che non si usa piu'. e JComponent, che e' figlia direttamente di Container (quindi non c'e' una Component in AWT), e che a sua volta ha diverse classi figlie, alcune riportate nel grafico. Notare che JPanel non e' una implementazione di Panel di AWT.

Dalla leggenda vediamo che Java definisce tre top-level container: JFrame, JDialog e JApplet. Un Container e' un Component che puo' contenere altri Component, se un Container e' top-level significa che rappresenta la radice della di una gerarchia di contenimento di certi Component di Swing. Ogni top-level container ha un **Content Pane** (nel Content Pane mettiamo i figli del top-level container; questo perche' il top-level container ha anche altro oltre al contenuto, es. tasto di chiusura, titolo: questa e' roba del top-level container che non va messa dentro il Content Pane) che ha i componenti visibili della GUI. Ogni applicazione che utilizza componenti Swing ha almeno un top-level container, precisamente ha almeno una gerarchia di contenimento con un JFrame come radice, poiche' un JFrame identifica una finestra con titolo e bordo dell'applicazione, e quindi e' fondamentale in Swing.

31 La Classe JFrame:

Iniziamo subito con un esempio:

```
import javax.swing.*;

public class GUIApp {
    public static void main(String[] args) {

        // Crea una finestra con titolo "Hello World"
        JFrame frm = new JFrame("Hello World!");

        // Si configura il frame in modo che l'applicazione termini
        // quando il pulsante di chiusura del frame viene premuto
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Rende la finestra visibile (di default non lo e')
        frm.setVisible(true);

        System.out.println("Ciao ciao");
    }
}
```

Quindi con `JFrame frm = new JFrame("Hello World!");` creiamo l'istanza frm della classe JFrame. Di default la finestra relativa a frm non e' visibile, quindi utilizziamo `frm.setVisible(true);` per renderla visibile, ma prima con `frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`³¹ forziamo la chiusura incondizionata del programma quando il pulsante di chiusura del frame frm viene premuto. Alla fine il metodo main stampa Ciao ciao. Ok, abbiamo gia' visto il costruttore di JFrame (c'e' anche la versione senza parametro che costruisce una finestra senza titolo), setVisible e setDefaultCloseOperation(int op), ecco gli altri metodi principali della classe JFrame:

³¹setDefaultCloseOperation() e' un metodo di JFrame che imposta il comportamento da adottare quando viene premuto il pulsante di chiusura di un frame. Ci sono 4 alternative: DO NOTHING ON CLOSE con cui il pulsante non fa nulla; HIDE ON CLOSE con cui la finestra si nasconde ma resta in memoria (riaccessibile con setVisible(true)); DISPOSE ON CLOSE che elimina la finestra liberando la memoria allocata e EXIT ON CLOSE che forza la chiusura del programma (equivalente a System.exit(0). Poiche' in questo corso identifichiamo un app grafica con il solo frame principale, alla chiusura del frame corrisponde la chiusura dell'app grafica, quindi per questo usiamo EXIT ON CLOSE. Se non lo andiamo a specificare, in Java il comportamento di default delle finestre e' HIDE ON CLOSE.

- `setTitle(String titolo)`: assegna *titolo* al titolo dell'oggetto di invocazione. Metodo ereditato dalla classe Frame di AWT.
- `void setSize(int lrg, int alt)`: assegna le dimensioni lrg e alt (specificate in pixels px) alla finestra oggetto di invocazione. Metodo ereditato dalla classe Component di AWT.
- `void setLocation(int x, int y)`: assegna posizioni date come parametri (coordinate specificate in pixels px) alla finestra oggetto di invocazione. L'origine e' in alto a sinistra. Metodo ereditato dalla classe Component di AWT.
- `void pack()`: ottimizza le dimensioni in base alle dimensioni preferite dai componenti contenuti nella finestra. Metodo ereditato dalla classe Window di AWT.
- `void setVisible(boolean b)`: gia' visto. Metodo ereditato dalla classe Component di AWT.

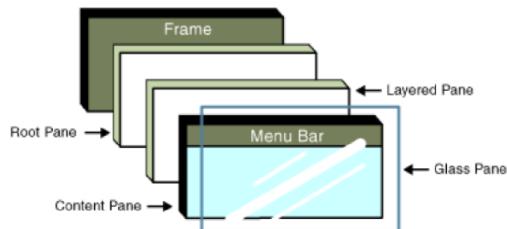
Altri metodi importanti, che vedremo dopo, sono:

- `Container getContentPane()`
- `void setContentPane(Container contentPane)`
- `void setJMenuBar(JMenuBar menuBar)`: permette di impostare un menu per la finestra.

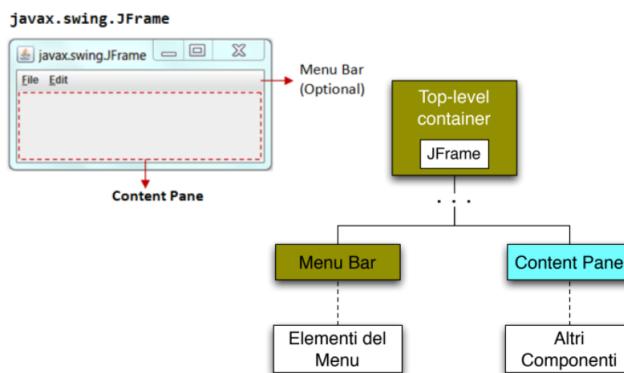
Gia' da questa divisione tra Menu e Content Pane capiamo che una finestra e' molto piu' di un Content Pane e basta: adesso siamo sicuri che una finestra oltre al Content Pane ha anche un menu.

31.1 Struttura a livelli di un oggetto JFrame:

Ciascuno strato svolge una funzione particolare, ma in questo corso noi consideriamo solo il Content Pane, poiche' gli altri strati permettono di implementare funzionalita' avanzate che non ci interessano. A che serve questo Content Pane? Ogni oggetto ha un livello Content Pane, acceden-



do ci possiamo creare finestre piu' ricche con pulsanti, aree di testo, caselle opzione ... Ecco qua



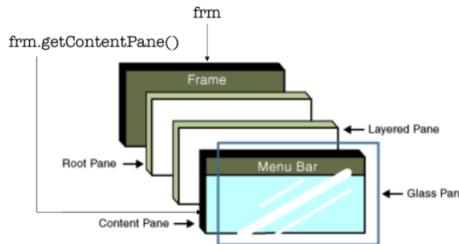
infatti che vediamo che una finestra Frame ha oltre che un Content Pane anche un Menu.

31.2 Popolare un JFrame:

Per popolare una finestra non possiamo semplicemente piazzare a caso come vogliamo i pulsanti e tutte le altre componenti grafiche, perche' c'e' una gerarchia di contenimento da rispettare? Ma perche' la gerarchia di contenimento, non ha piu' senso avere pulsanti, bottoni, menu ecc. tutti con la stessa "dignita'", cioe' dello stesso livello? No, conviene avere una gerarchia di contenimento con una radice top-level container perche' in questa maniera non devo andare a specificare per ogni componente grafica la sua posizione singolarmente. Quindi l'approccio per la creazione di una GUI in Java Swing e' con una gerarchia di contenimento ad albero, come in HTML. Popolare un oggetto JFrame significa aggiungere alla finestra corrispondente nuovi componenti grafici (bottoni, aree di testo ecc.). Ognuno di questi componenti grafici, per essere visibile sullo schermo, deve far parte di una gerarchia di contenimento, cioe' un albero di componenti che ha come radice un top-level container; ogni top-level container ha un Content Pane che contiene (direttamente o indirettamente) le componenti grafiche visibili. Quindi se vogliamo popolare JFrame abbiamo bisogno di un riferimento al suo Content Pane, per questo usiamo il metodo:

```
getContentPane()
```

Della classe JFrame. Questo metodo restituisce il riferimento al Content Pane, che e' di tipo Container (AWT). Possiamo poi popolare l'oggetto aggiungendo elementi al Content Pane tramite



il metodo:

```
Component add(Component comp)
```

Notare che questo, e altri metodi che utilizzeremo, appartiene a Component, che e' un'interfaccia di AWT: ogni volta che useremo un metodo o una variabile che e' istanza di una classe di AWT in realta' sara' istanza di una piu' specifica classe di Swing, e la useremo cosi'. Comunque, questo metodo aggiunge il Component comp al Content Pane oggetto di invocazione. Quindi il nostro esempio precedente puo' essere aggiornato cosi':

```
import javax.swing.*;
// Importa AWT per usare la classe Container
import java.awt.*;

public class GUIApp {
    public static void main(String[] args) {
        JFrame frm = new JFrame("Hello World!");
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Ottieni il riferimento al Content Pane
        Container frmContentPane = frm.getContentPane();

        // Usa frmContentPane per aggiungere elementi grafici
        frmContentPane.add(new JLabel("Buona Lezione!")); //una label di testo e' un'area di
        // testo su cui noi scriviamo ma che non sara' editabile dall'utente

        frm.setVisible(true);
    }
}
```

Nota che le dimensioni sono quelle di default ed il titolo non e' contenuto per intero. Proviamo a ingrandire le dimensioni di questa finestra:

```
import javax.swing.*;
```



```

import java.awt.*; //dobbiamo importarla perche' usiamo una variabile Container
(l'abbiamo chiamata frmContentPane)

public class GUIApp {
    public static void main(String args[]) {

        JFrame frm = new JFrame("Prima finestra");
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Imposta la dimensione del frame
        frm.setSize(200,200);

        Container frmContentPane = frm.getContentPane();
        frmContentPane.add(new JLabel("Buona Lezione"));
        frm.setVisible(true);
    }
}

```



Molto meglio. Concludendo, quindi il discorso e' che se devi modificare il menu hai bisogno di un riferimento (che e' una variabile) al menu su cui invocare i metodi per modificarlo; se devi modificare il Content Pane hai biosgno di un riferimento al Content Pane su cui invocare i metodi (e questo riferimento lo ottieni con getContentPane()); se devi modificare dimensione, altezza ecc. dell'intera finestra allora devi avere un'istanza della finestra (cioe' della classe) su cui invocare i metodi.

31.3 Modularita' delle GUI:

Fino ad ora abbiamo fatto gestire al modulo main() tutti gli aspetti dell'applicazione grafica. e' preferibile in realtu' avere invece una classe per ogni finestra, cosi' da delegare ad ognuna di queste classi le operazioni interne della finestra, cioe' le operazioni riguardanti dettagli implementativi della finestra (costruzione componenti grafici, creazione gerarchie di contenimento ecc.). Queste classi che modularizzano l'applicazione grafica devono estendere JFrame, cosi' da poter avere le funzionalita' di quest'ultima. Sempre riprendendo il nostro esempio di prima, aggiornandolo con la modularizzazione diventa:

```

//File MyFrame.java

import javax.swing.*; // AWT non serve (perche' non abbiamo usato nessuna variabile di
                     tipo Container, invece prima si'). In generale questo e' il metodo che si usa: non si
                     memorizza mai getContentPane in una variabile di tipo Container ma si fa direttamente
                     add su getContentPane

```

```

public final class MyFrame extends JFrame { //classe final ricordiamo che impone che
    questa classe non possa avere classi derivate

    private static final String titolo = "Prima Finestra";
    private final JLabel testo = new JLabel("Buona Lezione");
    private static final int larghezza=200, altezza=200;

    public MyFrame() {
        super(titolo); //poiche' MyFrame estende JFrame, chiamiamo il costruttore di
                      //JFrame classe madre per fargli mettere il titolo, che e' un campo dati
                      //ereditato, mentre quelli peculiari di MyFrame li facciamo in questo
                      //costruttore ovviamente
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(larghezza,altezza);
        this.getContentPane().add(testo);
        // Abbreviato: evitiamo il Container
        this.setVisible(true);
    }
}

```

```

//File GUIApp.java

public class GUIApp {
    public static void main(String[] args) {
        MyFrame myFrm = new MyFrame(); //qua si apre la finestra e il thread non termina
                                      //perche' il programma resta in attesa dell'input dell'utente nella finestra
    }
}

```

32 Layout Managers:

Gli elementi in un container sono disposti in una certa maniera. La maniera di default e' definita dal *Layout Manager* del container stesso. Ma cos'e' un Layout Manager? Si tratta di un oggetto che implementa l'interfaccia `LayoutManager`, e che determina dimensioni e posizione delle componenti all'interno di un Container. Per assegnare ad un Container (vedremo che ha senso farlo solo per Content Pane e JPanel) un particolare `LayoutManager` diverso da quello di default la classe Container ha il metodo:

```
void setLayout(LayoutManager manager)
```

Esistono tante implementazioni predefinite di `LayoutManager`, ognuna con una politica di dimensionamento e pozionamento dei componenti grafici nel container diversa, noi ci concentriamo su queste tre:

32.1 Classe FlowLayout:

Questa classe implementa un layout manager che dispone i componenti grafici riga per riga da sinistra verso destra. Non si va a caso finche' c'e' sufficiente spazio. I suoi due costruttori principali sono:

- `FlowLayout()` che crea un `FlowLayout` con allineamento centrale
- `FlowLayout(int align)` che crea un `FlowLayout` con allineamento specificato dal parametro `align`: `FlowLayout.LEADING` allinea a sinistra; `FlowLayout.CENTER` allinea al centro; `FlowLayout.TRAILING` allinea a destra.

Facciamo un esempio:

```

public final class MyFrame extends JFrame {

    private static final String titolo = "FlowLayout e JButton";
    private static final int larghezza=300, altezza=100;

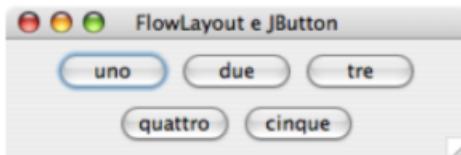
    private final JButton uno = new JButton("uno");
    // ... due, tre, quattro, cinque

    public MyFrame() {

        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(larghezza,altezza);
        Container frmContentPane = this.getContentPane();
        frmContentPane.setLayout(new FlowLayout()); //non specificando nulla
            l'allineamento sara' al centro
        frmContentPane.add(uno);
        // ... due, tre, quattro ...
        frmContentPane.add(cinque);
        this.setVisible(true);
    }
}

```

Quindi in Java, come le etichette - già viste - sono istanze della classe JLabel, i pulsanti sono istanze della classe JButton.



32.2 Classe GridLayout:

Questa classe implementa un layout manager che dispone i componenti grafici secondo una griglia di celle riga per riga da sinistra verso destra. Non si va a caso finché c'è sufficiente spazio. Ogni componente riempie interamente lo spazio della sua cella e tutte le celle hanno la stessa dimensione, determinata dal componente di dimensione massima. Inoltre le celle occupano tutto lo spazio disponibile nel container. Il costruttore principale della classe è:

- GridLayout(int *righe*, int *colonne*) che crea un GridLayout con *righe* righe e *colonne* colonne.
Il numero di celle sarà quindi $\text{righe} \times \text{colonne}$

Facciamo un esempio:

```

public final class MyFrame extends JFrame {

    private static final String titolo = "GridLayout e JButton";
    private static final int larghezza=200, altezza=200;
    private static final int righe=4, colonne=4;
    public MyFrame() {
        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(larghezza,altezza);
        Container frmContentPane = this.getContentPane();
        frmContentPane.setLayout(new GridLayout(righe, colonne)); //imposta la griglia a
            4x4
        for (int i = 0; i<15; i++) {
            frmContentPane.add(new JButton(String.valueOf(i))); //mette i numeri nei pulsanti
        }
        this.setVisible(true);
    }
}

```

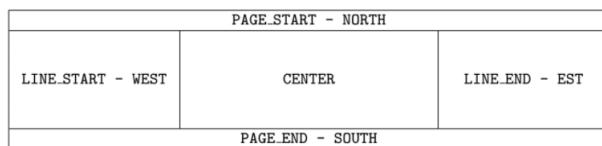
}

Che produce la seguente finestra:



32.3 Classe BorderLayout:

Questa classe implementa un layout manager che dispone i componenti grafici suddividendo il container in cinque aree: Il costruttore principale di questa classe è:



- BorderLayout() che crea un layout BorderLayout.

Un container con questo layout manager ha il metodo:

```
void add(Component c, String pos)
```

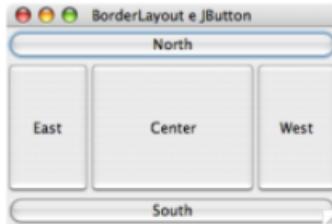
che consente di aggiungere componenti specificando in quale delle 5 posizioni:

- BorderLayout.CENTER
- BorderLayout.PAGE_START (o alternativamente BorderLayout.NORTH)
- BorderLayout.PAGE_END (o alternativamente BorderLayout.SOUTH)
- BorderLayout.LINE_START (o alternativamente BorderLayout.WEST)
- BorderLayout.LINE-END (o alternativamente BorderLayout.EAST)

Da considerare è anche il fatto che *aree senza elementi non vengono visualizzate e la finestra si adatta di conseguenza*. Facciamo un esempio:

```
public final class MyFrame extends JFrame {  
    private static final String titolo = "BorderLayout e JButton";  
    private static final int larghezza = 300, altezza = 200;  
    private final JButton north = new JButton("North");  
    // south, east, ...  
  
    public MyFrame() {  
        super(titolo);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        this.setSize(larghezza,altezza);  
        Container frmContentPane = this.getContentPane();  
        //ha borderlayout di default  
        frmContentPane.add(north, BorderLayout.PAGE_START);  
        frmContentPane.add(south, BorderLayout.PAGE_END);  
        // south, east, etc...  
        this.setVisible(true);  
    }  
}
```

Che produce la seguente finestra:



Potevamo anche prendere dei JPanel e metterli nelle 5 posizioni, e poi metterci i bottoni dentro (in realta' puoi anche prima mettere i buttoni nelle 5 posizioni e poi mettere i JPanel), cosi':

```
public final class MyFrame extends JFrame {
    private static final String titolo = "Titolo Finestra";
    private static final int larghezza = 400, altezza = 200;
    private final JButton north = new JButton("North");
    // south, east, ...
    private final JPanel northP = new JPanel();
    // southP, eastP, ...

    public MyFrame() {
        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(larghezza,altezza);
        Container frmContentPane = this.getContentPane();
        //ha borderlayout di default
        frmContentPane.add(northP, BorderLayout.PAGE_START);
        frmContentPane.add(southP, BorderLayout.PAGE_END);
        // eastP, etc...

        northP.add(north) //aggiunge il bottone
        southP.add(south)
        //east, west

        this.setVisible(true);
    }
}
```

Che produce la seguente finestra:

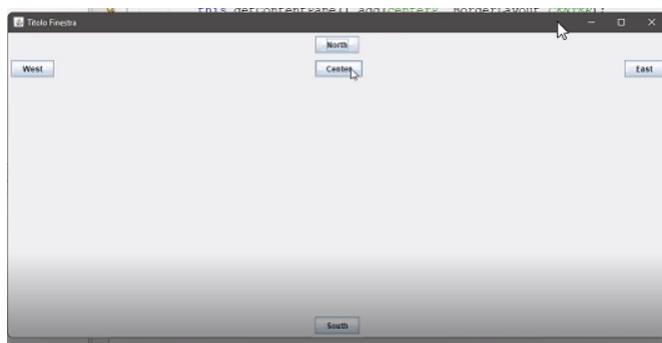


Figure 11: Non si badi al cambio di look and feel. L'importante e' notare che, dato che JPanel ha come Layout Manager di default Flow Layout, i 5 JButton vengono creati della grandezza corretta rispetto al testo

In genere, gli unici container per cui puo' essere necessario impostare un layout manager diverso da quello di default sono i Content Pane (che di default hanno layout BorderLayout) e i componenti

JPanel³² (che di default hanno layout FlowLayout).

33 Progettazione della GUI:

33.1 Container Annidati e JPanel:

Per creare interfacce grafiche complesse si possono comporre container diversi, seguendo una gerarchia di contenimento. Ogni container avra' un suo LayoutManager che disporra' in un certo modo gli elementi del container. Per fare questo usiamo i JPanel, che abbiamo gia' visto prima. Un JPanel e' un JComponent, quindi puo' essere aggiunto al content pane di un JFrame; ma allo stesso tempo sappiamo che e' un oggetto di tipo Container, quindi puo' contenere componenti grafici e anche altri JPanel. Un JPanel ha un proprio LayoutManager (quello di default e' FlowLayout). Il nostro obiettivo sara' proprio quello di costruire interfacce grafiche piu' complesse mettendo dei JPanel sulla finestra principale e su ognuno di questi panel useremo il layout che preferiamo.

Passi da seguire nella progettazione e nella realizzazione di un'interfaccia: per la progettazione si adotta un approccio **top-down**, cioe' partendo dal contenitore principale si procede verso i contnitori piu' interni (quindi dall'alto al basso della gerarchia di contenimento). I passi sono i seguenti:

- Si assegna un LayoutManager al ContentPane della finestra (di default BorderLayout)
- Per ciascuna area che si vuole aggiungere si crea un JPanel assegnando ad ognuno di essi un layout (di default FlowLayout)
- Ciascun pannello potra' a sua volta contenere altri JPanel o direttamente componenti grafici come JButton e JLabel.

Vediamo subito un primo esempio, come costruire la seguente finestra?



Progettazione (top-down): cioe' che viene pensato durante la fase di progettazione viene descritto da un albero della GUI (quindi una gerarchia di contenimento), con eventuali commenti per la successiva fase di realizzazione. Questa gerarchia si puo' fare a mano o con software di mockup appropriati, come Balsamiq. Nel nostro caso, il progetto ha tirato fuori la seguente gerarchia: Quindi in pratica l'idea sarebbe adottare un BorderLayout per il ContentPane del JFrame (cioe' dell'intera finestra), e poi:

- Per la zona nord mettiamo un JPanel con solo una JLabel dentro, quindi ci basta un BorderLayout
- Per la zona centrale mettiamo un JPanel con due JCheckBox per la scelta di un'opzione. Per fare cio' possiamo ad esempio scegliere per il JPanel il GridLayout con 1 colonna e 2 righe.
- Per la zona sub mettiamo un JPanel con FlowLayout, in modo che i due bottoni vengano messi uno accanto all'altro.

³²Un JPanel e' un componente che fa da container per altri componenti grafici (come bottoni, aree di testo ecc.)

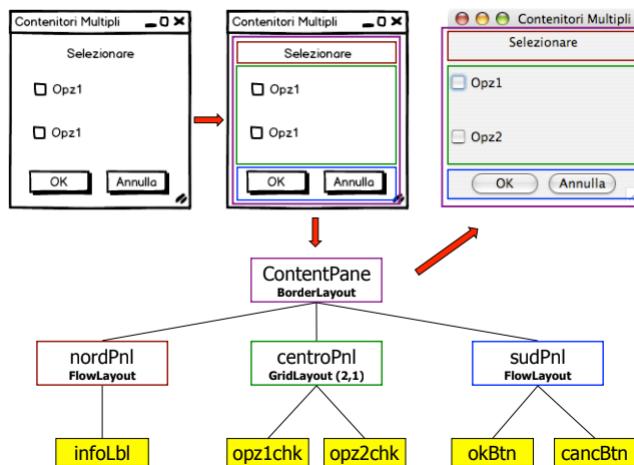


Figure 12: Partendo da sinistra, i primi due disegni sono appunto dei mockup, mentre la terza e' la vera e propria interfaccia grafica, che noi non sappiamo ancora come e' fatta, non avendo ancora realizzato il programma. Vediamo sotto la gerarchia di progettazione che indica bottoni, pannelli e opzioni senza la sintassi di Java, solo per far capire di che si parla.

Realizzazione (Bottom-up): arriviamo ora alla realizzazione in codice della nostra GUI, per la realizzazione si adotta il procedimento opposto rispetto a quello adottato nella progettazione, cioe' un approccio **bottom-up**: prima si instanziano le componenti grafiche atomiche (le foglie dell'albero di contenimento), poi i contenitori subito sopra, e cosi' via salendo. Vediamo quindi il vero e proprio codice della nostra finestra:

```

public final class MyFrame extends JFrame {

    private static final String titolo = "Contenitori Multipli";

    //Pannello Nord:
    private final JLabel infoLbl = new JLabel("Selezionare");
    private final JPanel nordPnl = new JPanel();

    //Pannello Centrale:
    private final JCheckBox opz1Chk = new JCheckBox("Opz1");
    private final JCheckBox opz2Chk = new JCheckBox("Opz2");
    private final JPanel centroPnl = new JPanel(new GridLayout(2,1));

    //Pannello Sud:
    private final JButton okBtn = new JButton("OK");
    private final JButton cancBtn = new JButton("Annulla");
    private final JPanel sudPnl = new JPanel();

    //Popoliamo i Container "dal basso verso l'alto" (nel costruttore)
    public MyFrame() {

        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Pannello Nord
        nordPnl.add(infoLbl);

        //Pannello Centro
        centroPnl.add(opz1Chk);
        centroPnl.add(opz2Chk);

        //Pannello Sud
        sudPnl.add(okBtn);
        sudPnl.add(cancBtn);
    }
}

```

```

//Container Principale
Container frmContentPane = this.getContentPane();
frmContentPane.add(nordPnl, BorderLayout.NORTH);
frmContentPane.add(centroPnl, BorderLayout.CENTER);
frmContentPane.add(sudPnl, BorderLayout.SOUTH);

//Impostiamo le proprieta' di visualizzazione: imposta la dimensione minima per
//visualizzare tutti i componenti
this.pack();

// Posizioniamo la finestra al centro dello schermo
this.setLocationRelativeTo(null);

// Rendiamo visibile la finestra
this.setVisible(true);
}
}

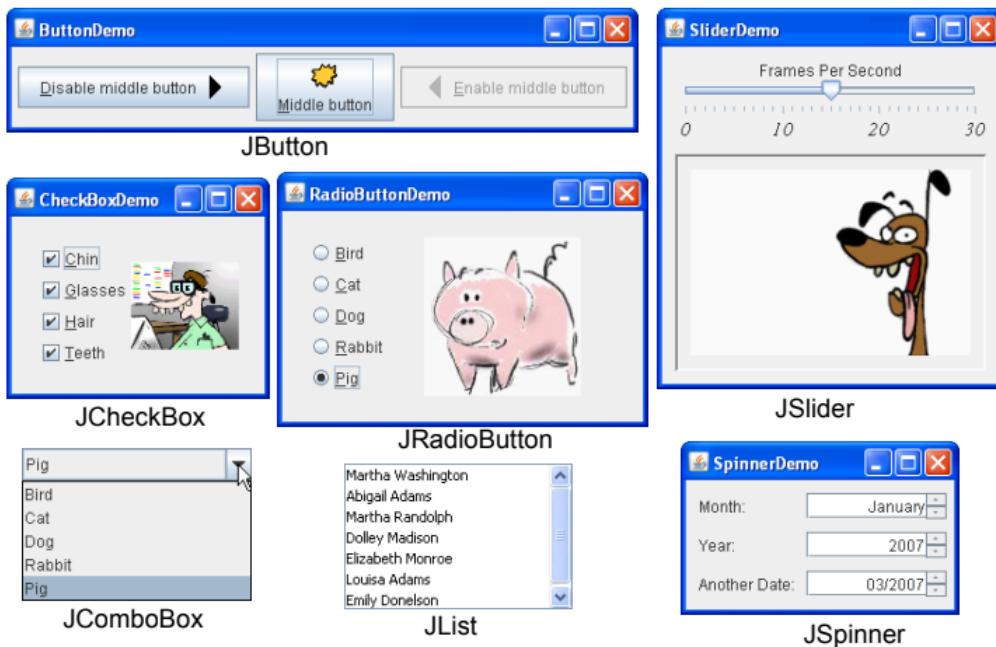
```

Osservazioni:

- Nel codice originale si utilizza static per JLabel, JPanel, JCheckBox ecc. ma il prof lo sconsiglia: la parola chiave static ci dice che cio' che e' static non e' una variabile di istanza ma di classe, quindi in tutto l'applicativo ci sara' una sola istanza di questa variabile. Cio' vuol dire che se io vado a creare istanze multiple della stessa finestra, avremo che queste istanze saranno delle finestre che usano gli stessi componenti grafici (es. static JButton vuol dire che due finestre della stessa classe usano lo stesso bottone); questo - a meno che non ci si assicuri di fare al massimo un'istanza per ogni JFrame in cui si dichiarano static queste variabili - puo' creare dei problemi, quindi meglio evitare. Invece e' consigliabile usare sempre e per tutte le variabili il qualificatore final, perche' questo porta dei vantaggi quando si utilizzano le inner-class (che vediamo dopo).
- Usiamo l'add sul ContentPane, come abbiamo sempre fatto9, con due parametri: il primo e' cosa effettivamente va aggiunto, il secondo e' la posizione. Questo e' ovviamente possibile nel caso in cui il ContentPane abbia come LayoutManager quello di default, cioe' BorderLayout. Questo pero' non causa nessun problema quando il LayoutManager e' un altro: in quel caso semplicemente il secondo parametro viene ignorato.
- Il metodo pack() decide la grandezza della nostra finestra in base a cio' che gli abbiamo messo dentro. Si tratta quindi di un metodo comodo, dato che non ci obbliga a settare una larghezza e un'altezza, ma a volte puo' causare problemi, e quindi diventa necessario settare le dimensioni a mano.

33.2 Alcuni JComponent Comuni:

Vediamo innanzitutto una figura esplicativa:



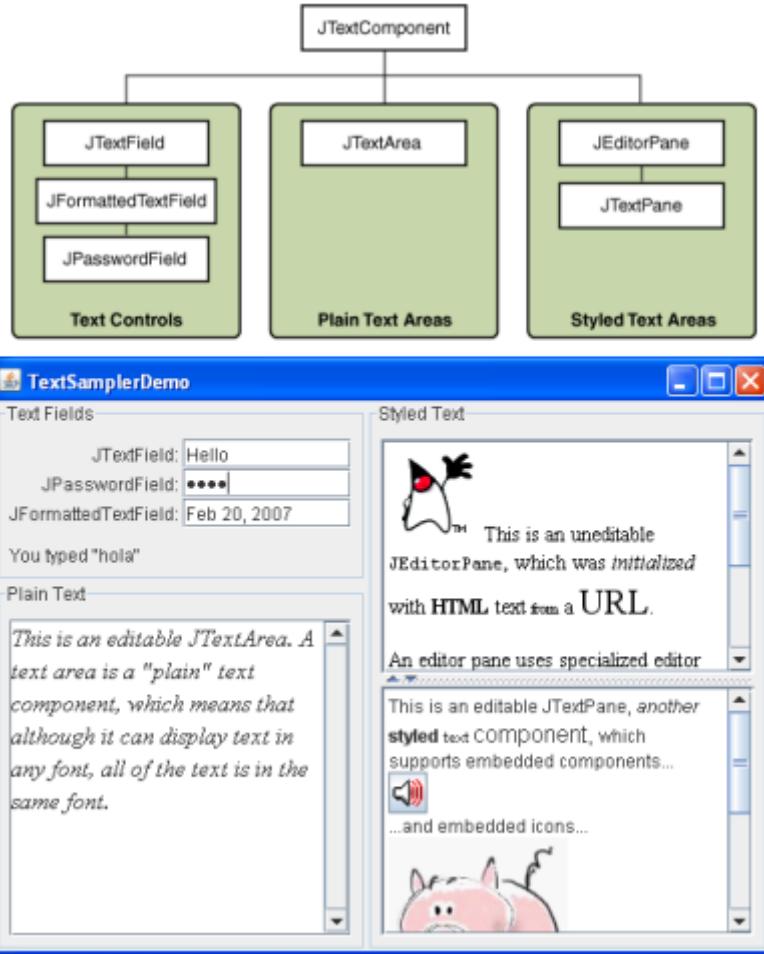
Ora vediamoli uno ad uno:

- JButton lo conosciamo.
- JCheckBox lo conosciamo.
- JRadioButton: trattasi di CheckBox a scelta esclusiva. Bisogna preferire il loro utilizzo in contenitori che definiscano gruppi di pulsanti. Questo perche' in un'intera finestra potresti aver bisogno di fare piu' scelte esclusive. Mettere semplicemente un JRadioButton non ti permetterebbe di fare piu' di una scelta per tutta la finestra, mentre avere dei gruppi di pulsanti ti permetterebbe di fare una sola scelta ma per contenitore, quindi diverse scelte nell'intera finestra.
- JSlider. Autoesplicativo.
- JSpinner: sono le frecce che vedi accanto al campo di testo. Permettono di scorrere tra i vari valori.
- JList: crea una lista di elementi, con la possibilita' di selezionarne piu' di uno.
- JComboBox: classica lista drop-down, clicchi ed escono le opzioni selezionabili.

33.2.1 JTextComponent (JComponent testuali):

Vediamo la gerarchia di JTextComponent e sotto una demo di cio' che si puo' fare:
Abbiamo:

- JTextField che e' una classica casella di testo.
- JFormattedTextField che e' una casella di tempo in cui e' impostato un formato e richiede agli utenti di rispettare quel formato di inserimento.
- JPasswordField che e' una casella di testo in cui non si possono vedere i caratteri e non si puo' fare copia.
- JTextArea e' la classica area di testo
- JEditorPane e JTextPane che sono compatti grafici avanzati, da cui e' possibile selezionare il font, gli effetti del testo ecc. La cosa che differenzia i due tra loro e' che uno dei due e' compatibile con la traduzione in HTML l'altro no (non si ricorda quale dei due).



34 I Menu: Progettazione e Realizzazione:

Come abbiamo già visto, un `JFrame`, cioè una finestra, è formata sia da un `ContentPane` che da una `JMenuBar`, cioè una barra menu, anche se questa può non esserci.

La barra menu: I menu strutturano gerarchicamente opzioni e funzionalità della finestra e dell'applicazione. Essi sono contenuti nella barra menu, posizionata tipicamente nella parte alta della finestra. Ogni finestra, quindi ogni `JFrame`, può contenere al massimo una barra menu. Come abbiamo `getContentPane()` per ottenere un riferimento al Content Pane della finestra, il costruttore `JMenuBar()` per costruire un riferimento `JMenuBar` alla barra menu. Dopo aver creato un riferimento a una barra menu bisogna assegnarlo a una finestra, cioè a un `JFrame`, tramite il metodo della classe `JFrame`:

```
void setJMenuBar(JMenuBar menubar)
```

I menu: Una barra menu, quindi un oggetto `JMenuBar`, può contenere un numero qualsiasi di oggetti `JMenu`, che sono riferimenti ai singoli menu. Per ottenere un riferimento a un singolo menu si utilizza il costruttore `JMenu(String nome)` che prende in input il nome del menu. Dopo aver creato un riferimento a un singolo menu bisogna assegnarlo a una barra menu, cioè a un `JMenuBar`, tramite il metodo della classe `JMenuBar`:

```
JMenu add(JMenu m)
```

L'ordine in cui vengono visualizzati i menu nella barra dei menu è l'ordine di utilizzo di `add`: prima aggiungi un menu con `add` e prima verrà visualizzato.

Sottomenu, Voci e Separatori di un menu: Un menu, quindi un oggetto `JMenu`, può contenere altri oggetti `JMenu`, quindi dei menu annidati, o sottomenu. Il comando per aggiungere un sottomenu è il seguente metodo di `JMenu`:

```
Component add(Component)
```

Oltre ai sottomenu i JMenu possono ovviamente contenere anche degli oggetti di tipo JMenuItem, cioe' le voci del menu. Per creare una voce si utilizza il seguente costruttore:

```
JMenuItem(String voce)
```

E per aggiungere la voce creata in un menu si utilizza il metodo della classe JMenu:

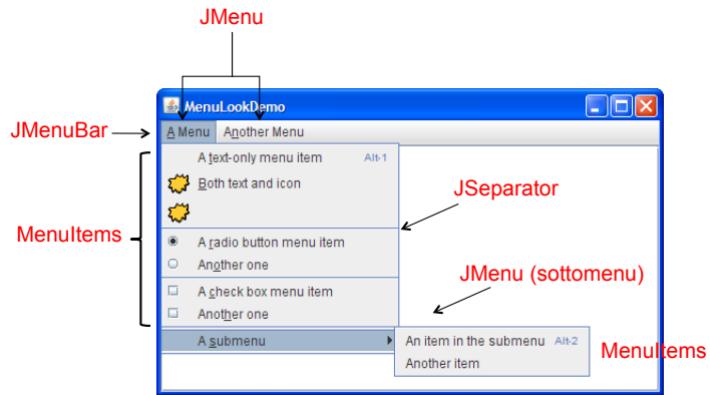
```
JMenu add(JMenuItem item)
```

Se vogliamo possiamo aggiungere dei separatori JSeparator tra le voci di un menu, con il comando di JMenu:

```
add(JSeparator separatore)
```

L'ordine in cui vengono visualizzati i sottomenu, i separatori e le voci nel menu e' l'ordine di utilizzo di add: prima aggiungi e prima verrà visualizzato.

Vediamo, graficamente, l'esempio di una finestra con una barra di menu, dei menu, dei sottomenu e delle voci, sia dei menu che dei sottomenu:

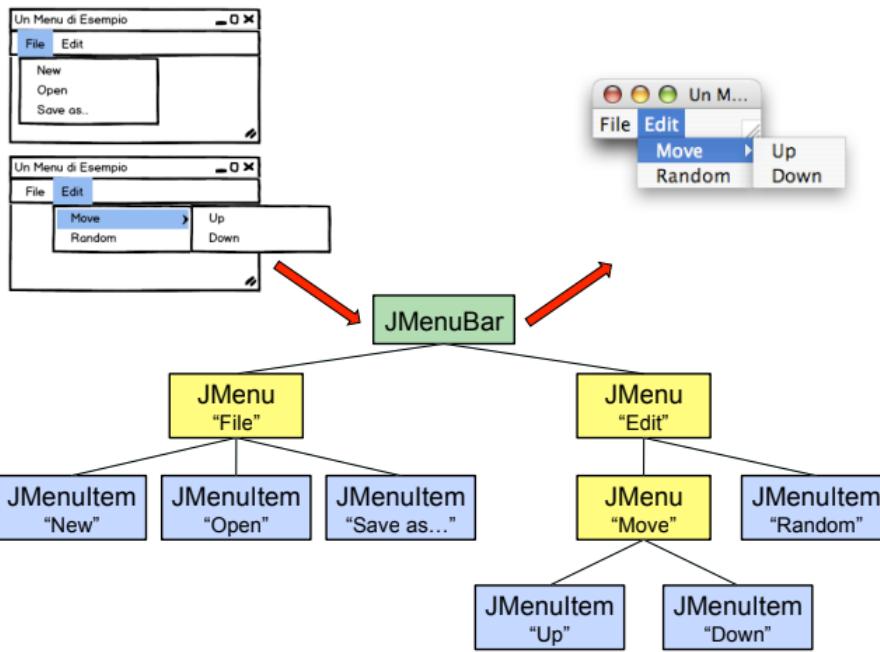


Vediamo dalla figura che e' quindi possibile andare a creare dei JMenuItem di tipo particolare (come JRadioButton o JCheckBox), che e' possibile aggiungere dei JSeparator (cioe' delle righe orizzontali nel menu) e che per ogni JMenuItem possiamo aggiungere degli shortcut da tastiera. Anche per i menu adottiamo un'approccio top-down alla progettazione e uno bottom-up alla realizzazione. Partiamo con la progettazione.

Progettazione dei menu (top-down): partiamo dalla barra menu, definiamo i singoli menu, poi i sottomenu e infine le voci di menu, cioe' le foglie della gerarchia di contenimento. Cioe' abbiamo queste tre fasi:

- Definire i JMenu della JMenuBar
- Per ogni JMenu distinguere i sottomenu JMenu e le voci di menu JMenuItem
- Per i sottomenu JMenu ripetere il punto precedente

Anche qui, come nella progettazione del ContentPane, che sia a mano o con SW, e' possibile disegnare il menu graficamente, per capire come lo vogliamo. Dopo averlo disegnato, e in base al disegno, possiamo fare la gerarchia, per capire come andrebbe realizzato. Esempio:



Realizzazione dei menu (bottom-up): al contrario, per realizzarlo il menu si procede dalle voci (le foglie della gerarchia), poi si istanziano i sottomenu, partendo da quelli piu' profondi, fino ad arrivare ai menu e infine alla barra menu. Per questo nostro esempio che stiamo vedendo, il codice realizzativo e' il seguente:

```

public class MyMenu extends JFrame {

    private static final String titolo = "Un Menu di Esempio";

    //Menu Move
    private static final JMenuItem upIt = new JMenuItem("Up");
    private static final JMenuItem dwnIt = new JMenuItem("Down");
    private static final JMenu moveMenu = new JMenu("Move");

    //Menu di Editing
    private static final JMenuItem rndmIt = new JMenuItem("Random");
    private static final JMenu editMenu = new JMenu("Edit");

    //Menu File
    private static final JMenuItem newIt = new JMenuItem("New");
    private static final JMenuItem openIt = new JMenuItem("Open");
    private static final JMenuItem saveIt = new JMenuItem("Save as...");
```

```
    private static final JMenu fileMenu = new JMenu("File");

    //Menu Bar
    private final JMenuBar menuBar = new JMenuBar();
```

```
// Popoliamo i menu "dal basso verso l'alto" (nel costruttore)
public MyMenu() {
```

```
    super(titolo);

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Move Menu
    moveMenu.add(upIt);
    moveMenu.add(dwnIt);

    // Edit Menu
    editMenu.add(moveMenu);
```

```

        editMenu.add(rndmIt);

        // File Menu
        fileMenu.add(newIt);
        fileMenu.add(openIt);
        fileMenu.add(saveIt);

        // Barra menu
        menuBar.add(fileMenu);
        menuBar.add(editMenu);

        // Imposta la Barra Menu nel frame
        this.setJMenuBar(menuBar);

        // Impostazioni di visualizzazione
        this.pack();
        this.setLocationRelativeTo(null);
        this.setVisible(true);

    }
}

```

35 Eventi e Ascoltatori:

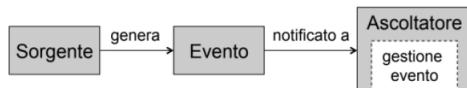
Finora abbiamo visto come progettare e realizzare interfacce grafiche Swing focalizzandoci solo sugli aspetti di visualizzazione. Ma, come è noto, molti componenti grafici consentono l'interazione con l'utente. È quindi possibile far corrispondere determinate azioni alla pressione di un bottone, al posizionamento del cursore o alla pressione di pulsanti del mouse o della tastiera. Questo viene chiamato modello event-driven, cioè guidato dall'occorrenza di eventi. Questi eventi, a cui l'applicazione reagisce durante l'esecuzione, possono essere generati dall'utente o da altre sorgenti (connessioni di rete, timers ecc.). In questo modello ad eventi distinguiamo:

- Sorgenti di Eventi: le componenti della finestra il cui stato cambia (il bottone quando viene premuto cambia stato e genera un evento).
- Eventi: oggetti che rappresentano e incapsulano le informazioni relative al cambio di stato delle sorgenti di eventi.
- Ascoltatori di Eventi: oggetti a cui viene notificato l'evento. Una volta ricevuta la notifica dell'evento avvenuto eseguono opportune azioni.

Questo modello ad eventi ed ascoltatori è un pattern standard che viene utilizzato anche al di fuori della progettazione GUI, e viene chiamato anche modello publisher-subscriber, oppure listener-listening.

35.1 Event Delegation:

Ogni volta che una sorgente genera un evento, l'evento generato viene notificato agli ascoltatori sottoscrittori (cioè interessati), che eseguono opportune azioni. Quindi di fatto la sorgente **delega** agli ascoltatori la gestione degli eventi che la sorgente stessa ha generato. Per implementare l'event



delegation bisogna associare ad ogni sorgente che ci interessa (quindi ad esempio ad ogni bottone), uno o più opportuni ascoltatori (event listener), a cui verrà delegata la gestione dell'evento. Ciascuna sorgente può avere più ascoltatori per uno stesso evento (quindi ogni sorgente gestisce una lista di ascoltatori), e più sorgenti possono avere uno stesso ascoltatore per lo stesso evento. Detto questo, vediamo in pratica come realizzare questa gestione degli eventi.

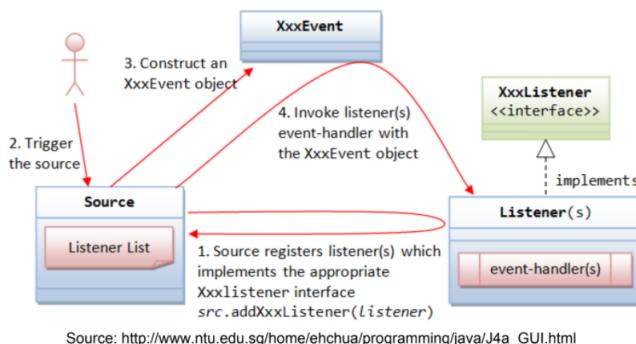


- Innanzitutto va identificato il tipo di eventi da gestire.
- Trovato questo tipo, bisogna definire una classe che implementi l'interfaccia Listener corrispondente al tipo scelto. L'interfaccia Listener avrà metodi che ricevono e gestiscono gli eventi (questi metodi sono quindi gli event handlers) del tipo identificato; la classe che implementa l'interfaccia specificherà la logica di gestione degli eventi relativa ad ogni singolo metodo.

in questo modo abbiamo realizzato il meccanismo, ora vediamo come funziona, cioè come avviene l'interazione tra utente e componenti:

- Innanzitutto registriamo un'istanza della classe che implementa l'interfaccia Listener presso la sorgente che genera gli eventi del tipo identificato. Cioè facciamo quello che prima abbiamo chiamato "associare a una sorgente uno o più opportuni ascoltatori (event listeners)". Ogni sorgente gestisce una lista di questi ascoltatori.
- L'utente interagisce con la sorgente
- L'evento viene notificato alla lista di ascoltatori registrati presso la sorgente. Gli event handler (metodi) degli ascoltatori vengono eseguiti come programmato.

Quindi graficamente possiamo schematizzare così il processo:



35.2 Facciamo un esempio con il Tipo di eventi MouseEvent:

All'interno del package `java.awt.event` è definita la classe `MouseEvent`. Questa classe definisce quindi il tipo di evento `MouseEvent`. Un oggetto di questa classe viene istanziato ogni volta che avviene un evento del mouse (o dispositivo simile, come il touchpad) rispetto a una certa sorgente scelta. Precisamente, avviene un evento di tipo `MouseEvent` quando:

- Il cursore del mouse entra o esce dall'area visibile occupata dalla sorgente
- L'utente preme, rilascia o clicca un pulsante nel componente (nel componente che è sorgente dell'evento si intende).

Come detto prima, l'evento è un oggetto che contiene informazioni sul cambio di stato della sorgente avvenuto. Le suddette informazioni incapsulate dall'evento sono accessibili tramite opportuni metodi, nel caso di `MouseEvent` abbiamo:

- `int getX()` e `int getY()` che restituiscono le coordinate cartesiane del mouse relative alla componente (sorgente) in cui l'evento è stato generato.
- `int getButton()` permette di conoscere quale pulsante del componente ha generato l'evento. Restituisce un `int` perché restituisce una delle seguenti costanti: `MouseEvent.NOBUTTON`, `MouseEvent.BUTTON1`, `MouseEvent.BUTTON2`, `MouseEvent.BUTTON3`. ...

- int getClickCount() che restituisce il numero di click associati all'evento.

Come abbiamo detto, ad un tipo di evento corrisponde un certo Listener. Infatti, al tipo di evento MouseEvent corrisponde l'interfaccia MouseListener (che si trova sempre nel package java.awt.event). Questo Listener definisce gli event handlers, cioe' i metodi per la gestione di eventi di tipo MouseEvent. Vediamo il codice sorgente dell'interfaccia MouseListener:

```
public interface MouseListener {

    // Invocato quando si clicca sul componente associato
    void mouseClicked(MouseEvent e);

    // Invocato quando il puntatore del
    // mouse entra nell'area del componente
    void mouseEntered(MouseEvent e);

    // Invocato quando il puntatore del
    // mouse esce dall'area del componente
    void mouseExited (MouseEvent e);

    // Invocato quando un pulsante del
    // mouse viene premuto sul componente
    void mousePressed(MouseEvent e);

    // Invocato quando un pulsante del
    // mouse viene rilasciato sul componente
    void mouseReleased(MouseEvent e);

}
```

Questa e' l'interfaccia che poi l'Ascoltatore (cioe' la classe di cui sono istanze i nostri ascoltatori) implementera' per acquisire i giusti metodi (event handlers) per gestire gli eventi del tipo MouseEvent³³ (ma poi siamo noi con la nostra implementazione che decidiamo cosa fanno questi metodi). Bene, facciamo allora questa classe ascoltratrice, che implementa MouseListener:

```
import java.awt.event.*;

public class MouseSpy implements MouseListener {

    public void mouseClicked(MouseEvent e) {
        System.out.println("Click su (" + e.getX() + "," + e.getY() + ")");
    }

    public void mousePressed(MouseEvent e) {
        System.out.println("Premuto su (" + e.getX() + "," + e.getY() + ")");
    }

    public void mouseReleased(MouseEvent e) {
        System.out.println("Rilasciato su (" + e.getX() + "," + e.getY() + ")");
    }

    public void mouseEntered(MouseEvent e) {} // evento non gestito
    public void mouseExited(MouseEvent e) {} // evento non gestito
}
```

Questi ascoltatori, cioe' gestori di eventi, sono associabili a qualsiasi JComponent, ma anche al Content Pane o all'intero JFrame. Infatti adesso facciamo un esempio associandolo all'intero JFrame della GUI:

```
import javax.swing.*;
```

³³Nota che MouseEvent e' diverso da MouseMotion perche' il secondo ha metodi che gestiscono gli eventi relativi a un mouse che si muove in una componente, mentre il primo ha metodi che gestiscono solo l'entrata e l'uscita da una componente. Quindi il secondo ha metodi che seguono il mouse continuamente diciamo.

```

public class MyFrame extends JFrame {

    public MyFrame() {

        super("MouseTest");

        // registriamo l'ascoltatore presso il frame
        this.addMouseListener(new MouseSpy());

        this.setSize(200,200);
        this.setVisible(true);
    }

    public static void main(String[] args) { //mettiamo il metodo main nella stessa
        classe ma e' cattiva pratica di programmazione
        new MyFrame();
    }
}

```

Presentiamo qui una tabella con i principali Eventi e le corrispettive interfacce Listener da implementare per gestire gli eventi di quel tipo, con una breve descrizione:

Evento	Interfaccia Listener	Descrizione interfaccia
ActionEvent	ActionListener	Definisce un metodo per gestire eventi di tipo "azione" (vedi slide successive)
ComponentEvent	ComponentListener	Definisce 4 metodi per riconoscere quando un componente viene nascosto, spostato, mostrato o ridimensionato
FocusEvent	FocusListener	Definisce 2 metodi per riconoscere quando un componente ottiene o perde il focus da tastiera
KeyEvent	KeyListener	Definisce 3 metodi per riconoscere quando un tasto (della tastiera) viene premuto, rilasciato o battuto
MouseEvent	MouseMotionListener	Definisce 2 metodi per riconoscere quando il puntatore del mouse è spostato o trascinato su un componente
MouseEvent	MouseListener	Gestisce gli eventi del mouse (vedi slide precedenti)
TextEvent	TextListener	Definisce un metodo per riconoscere quando il valore di un campo testuale cambia
WindowEvent	WindowListener	Definisce 7 metodi per riconoscere quando una finestra viene attivata, eliminata, chiusa, disattivata, ripristinata, ridotta a icona, o resa visibile

Concentriamoci sui primi, cioe' gli ActionEvent e ActionListener. Gli eventi di tipo ActionEvent vengono generati in corrispondenza di **Azioni**. Un'azione e' un comando che l'utente da' a un componente. Ad esempio le azioni possono essere:

- Click su un bottone
- Selezione di una voce in un menu
- Pressione del tasto "Invio" in un campo di testo
- Selezione di un elemento in una lista combo box
- Selezione di un radio button
- ...

Notiamo quindi che anche ActionEvent ha i click, come MouseEvent. Il discorso e' che se un pulsante noi lo dobbiamo premere, in qualsiasi modo ma l'importante e' che lo premiamo, allora e' consigliato usare ActionEvent e NON MouseEvent, questo perche' ActionEvent definisce come azione qualsiasi cosa che permetta la pressione del tasto: non solo il click del mouse ma, utilizzando tab per spostarsi tra i controlli di una finestra, anche la pressione del tasto invio³⁴ per premere un bottone, ad esempio. Mentre se per qualche motivo tu vuoi che l'evento si scateni SOLO se

³⁴ATTENZIONE: in Java il tasto in realta' non e' invio ma e' il tasto barra spaziatrice! Quindi l'ActionEvent considera un pulsante premuto se, ottenuto il focus da tastiera su di lui, si preme il tasto barra spaziatrice, non il tasto invio! Questo perche' Java riserva il tasto invio al pulsante di default della finestra: cioe', quando una finestra si apre, viene selezionato il focus su un pulsante di default, quel pulsante puo' essere subito premuto premendo il

premi col mouse il bottone (e non ad esempio se premi il bottone con invio) allora devi usare MouseEvent. Questo e' il tipo ActionEvent, poi abbiamo la sua corrispondente interfaccia, ovvero ActionListener (definita sempre nel package java.awt.event). Come gia' detto nella tabella, questa interfaccia definisce un solo event handlers, cioe' metodo, per gestire gli eventi del tipo ActionEvent. Il suo codice e' infatti questo:

```
public interface ActionListener {  
  
    void actionPerformed(ActionEvent e);  
  
}
```

Il procedimento di gestione degli eventi di tipo ActionEvent segue lo schema generale definito prima. Quindi:

- Definiamo una classe che implementa l'interfaccia ActionListener. Questa classe eredita l'unico metodo dell'interfaccia ActionListener e lo specifica, decidendo qual e' la logica di gestione dell'evento.
- Registriamo un'istanza di questa classe come Ascoltatore (Listener) per il componente sorgente dell'evento, utilizzando il metodo:

```
void addActionListener(ActionListener l)
```

Questo metodo viene invocato sul componente sorgente dell'evento.

- Quando l'utente interagisce con il componente, questo genera un evento, cioe' un oggetto ActionEvent. Questa generazione dell'evento viene notificata all'istanza dell'ascoltatore registrato presso il componente sorgente. Questi ascoltatori allora invocano il metodo ActionPerformed.

Questo e' il modo di realizzare una gestione di eventi ActionEvent. Facciamo un esempio per capire meglio:



Vogliamo che, quando viene premuto uno dei cinque bottoni, si apra una finestra che ci dica quale bottone e' stato premuto. Per farlo, come gia' detto, creiamo una classe ascoltratrice che implementi ActionListener:

```
import java.awt.event.*;  
import javax.swing.*;  
  
public class MyFrameListener implements ActionListener {  
  
    public void actionPerformed(ActionEvent e) {  
  
        // nota che supponiamo che questo listener venga usato solo con i JButton, infatti  
        // dal primo comando castiamo e assegniamo a JButton  
  
        // getSource() restituisce un riferimento all'oggetto che ha generato l'evento  
        JButton b = (JButton) e.getSource();  
  
        // getText() di JButton restituisce il testo definito come etichetta del bottone
```

tasto invio da tastiera (es nelle message box Ok e' il pulsante di default, e premendo invio premiamo ok). Tutti gli altri pulsanti della finestra possono essere premuti, dopo aver ottenuto il focus da tastiera su di loro grazie al tasto tab, proprio quindi con la barra spaziatrice.

```

        JOptionPane.showMessageDialog(null,"Hai premuto il pulsante: "+ b.getText());
        //showMessageDialog prende come primo parametro la finestra genitore della Dialog.
        Questo significa che non si puo' usare la finestra genitore finche' non si
        chiude il Dialog. Mettendo null non c'e' nessuna finestra genitore, e il
        comportamento della finestra dialog in questo caso e' quello di BLOCCARE OGNI
        FINESTRA SOTTOSTANTE, quindi non solo il genitore, ma proprio tutte tutte.
    }
}

```

Ora associamo un'istanza di questa classe, quindi un ascoltatore, a ciascun pulsante definito nell'interfaccia grafica. Percio' nella classe MyFrame, che rappresenta la nostra intera finestra, abbiamo:

```

public final class MyFrame extends JFrame {

    // private static final String titolo, larghezza, altezza ...

    private final JButton uno = new JButton("uno");
    // due, ... , cinque

    // Creo un'istanza di MyFrameListener, cioe' un ascoltatore
    private static final MyFrameListener listener = new MyFrameListener();

    public MyFrame() {

        super(titolo);

        //chiusura, dimensioni, layout, content pane ...

        // Associa il listener al pulsante uno
        uno.addActionListener(listener);
        // idem per due, tre, quattro, cinque
        // Cioe' abbiamo associato lo stesso listener a 5 pulsanti diversi

        this.getContentPane().add(uno);
        // idem per due, tre, quattro, cinque

        this.setVisible(true);
    }
}

```

Volendo fare qualcosa di piu' complesso:

```

package provaction;

import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;

public class MyFrame extends JFrame {

    //Campi dato dell'intero Frame

    private static final String titolo = "Prova Action";

    //Content Pane

```

```

Container contentPaneThis = this.getContentPane();

private final JButton[] bottoni; //vogliamo che sia l'utente a inserire
// a tempo di esecuzione il numero di bottoni

    //ascoltatore
private final MyFrameListener ascoltatore = new MyFrameListener();

//Barra Menu

JMenuBar barramenu = new JMenuBar();

JMenu menuFile = new JMenu("File");
JMenuItem salva = new JMenuItem("salva");
JMenuItem salvaConNome = new JMenuItem("salva con nome");
JMenuItem bestemmia = new JMenuItem("bestemmia");

JMenu menuEdit = new JMenu("Edit");
JMenuItem rimuovi = new JMenuItem("rimuovi");

JMenu muovi = new JMenu("muovi");
JMenuItem destra = new JMenuItem("destra");
JMenuItem sinistra = new JMenuItem("sinistra");

public MyFrame(int numBottoni) { //quando faccio una finestra chiedo il numero di
    pulsanti

    super(titolo);

    bottoni = new JButton[numBottoni]; //inizializziamo a tempo di esecuzione l'array
    //perche' a tempo di compilazione il numero di bottoni non e' noto

    //Settore Content Pane
    contentPaneThis.setLayout(new FlowLayout());

    for (int i = 0; i < numBottoni; i++) {

        int numero = i + 1;
        bottoni[i] = new JButton("n. " + numero + " ");

        bottoni[i].addActionListener(ascoltatore);
        contentPaneThis.add(bottoni[i]);
    }

    //Settore Barra Menu

    this.setJMenuBar(barramenu);

    muovi.add(destra);
    muovi.add(sinistra);

    menuEdit.add(rimuovi);
    menuEdit.add(muovi);
}

```

```

        menuFile.add(salva);
        menuFile.add(salvaConNome);
        menuFile.add(bestemmia);

        barramenu.add(menuFile);
        barramenu.add(menuEdit);

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.pack();
        this.setVisible(true);
    }

    public static void main(String[] args) {
        MyFrame primaFinestra = new MyFrame(Integer.parseInt(args[0]));
        //quando si esegue il programma l'utente inserisce la stringa che, fatto
        //il parse int, rappresenta il numero di buttoni che si vuole nella finestra
        //sappiamo che da riga di comando gli argomenti in input al main
        //vengono inseriti accanto al nome del file da eseguire, mentre su NetBeans
        //bisogna fare cosi': tasto destro sul progetto -> properties -> run -> arguments

        //NOTA: SE AGGIUNGI DEGLI ARGOMENTI IN NETBEANS, DEVI SETTARE UNA CLASSE MAIN
        //(sempre in progetto -> properties -> run) E RUNNARE L'INTERO PROGETTO
        //NON IL SINGOLO FILE COME FACCIAMO DI SOLITO: per runnare l'intero progetto
        //o premi run sul simbolo del caffè' (quindi sull'intero progetto e non sul singolo
        //file) oppure premi il tasto verde che sembra un tasto play nella barra in alto
    }

}

-----  

package provaction;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JOptionPane;

public class MyFrameListener implements ActionListener {

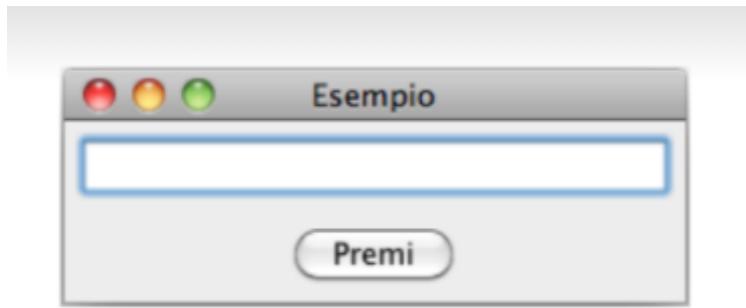
    @Override
    public void actionPerformed(ActionEvent e) {
        JButton sorgente = (JButton) e.getSource();

        String etichettaSorgente = sorgente.getText();
        JOptionPane.showMessageDialog(null, "Si e' premuto il tasto " + etichettaSorgente);
    }
}

```

Ora qualcosa di piu' difficile: Vogliamo costruire la seguente finestra: Per quanto riguarda la visualizzazione nulla di nuovo, il codice e' il seguente:

```
public class EsempioTextField extends JFrame {
```



```

private static final String titolo = "Esempio";
private static final JPanel centro = new JPanel();
private static final JPanel sud = new JPanel();
private static final JTextField campoTesto = new JTextField(20); //20 sara' il
    numero massimo di colonne (cioe' di caratteri) visibili insieme: se scriverai una
    stringa piu' lunga di 20 caratteri non potrai vederla tutta assieme ma dovrà
    scorrere
private static final JButton button = new JButton("Premi");

public EsempioTextField() {

    super(titolo);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    centro.add(campoTesto);
    sud.add(button);
    this.getContentPane().add(centro, BorderLayout.CENTER);
    this.getContentPane().add(sud, BorderLayout.SOUTH);

    button.addActionListener(new EsempioTextFieldListener());
    this.pack();
    this.setVisible(true);
}

}

```

Ok ora occupiamoci di quello che deve fare questa finestra. Vogliamo che, al click del bottone "premi", compaia una finestra che stampi cio' che abbiamo scritto nella casella di testo. Nota che questo non e' semplice come prima: non possiamo usare getSource() ora perche' non vogliamo semplicemente prendere il testo del bottone che premiamo, ma vogliamo prendere il testo di un altro componente della finestra. Ci sono diversi modi per farlo:

- Listener come inner class (SCONSIGLIATA³⁵). Ma che e' una **inner-class**? Una inner-class e' una classe definita all'interno di un'altra classe.

```

public class EsempioTextField extends JFrame {

    ...

    //inner class
    private class EsempioTextFieldListener implements ActionListener {

        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null, campoTesto.getText());
        }
    }
}

```

³⁵anche se la maggior parte del codice che si trova online e' scritto seguendo questo metodo, i professori lo considerano poco pulito.

}

Vediamo che in questo modo possiamo accedere direttamente ai componenti del frame, perche' le variabili della classe sono private ma noi siamo nella stessa classe; quindi, contestualmente all'uso delle variabili di classe, un'inner-class si comporta come un metodo di classe. Si tratta come gia' detto di un metodo non pulito, perche' mischia il codice del Frame col codice di gestione degli eventi. Qualche versione fa di Java era obbligatorio qualificare final le variabili di classe corrispondenti alle componenti del frame, altrimenti non erano accessibili dall'inner class.

- Listener implementato direttamente dalla classe che estende JFrame:

```
public class EsempioTextField extends JFrame implements ActionListener {

    // Definizione di: centro, sud, campoTesto, button, titolo (come sopra)
    private static final JTextField campoTesto = new JTextField(20);

    public EsempioTextArea() {

        // Creazione e impostazione finestra (come sopra)
        // il listener e' implementato dalla classe stessa
        button.addActionListener(this); //passo me stesso come listener
        this.pack();
        this.setVisible(true);
    }

    // Implementazione di actionPerformed per la gestione degli eventi
    public void actionPerformed(ActionEvent e) {
        // Accede direttamente al componente "campoTesto"
        JOptionPane.showMessageDialog(null, campoTesto.getText());
    }
}
```

Anche qui siamo in grado di accedere direttamente a tutti i campi dato della classe, anche se sono privati. Notare che anche questo modo ha il problema di non essere pulito poiche' mischia il codice della gestione degli eventi e il codice del frame. In ogni caso comunque e' meglio del modo precedente.

- Listener come classe esterna (come abbiamo fatto fin'ora). Se usiamo questo metodo dobbiamo passare alla classe esterna un riferimento alla finestra che contiene il componente sorgente che ci interessa. In questo modo il listener ha accesso ha tutti i campi PUBBLICI della nostra finestra, e quindi nella classe della nostra finestra scriviamo dei metodi getter pubblici che ci permettono di ottenere le variabili (cioe' le componenti) private. Nel nostro caso dobbiamo dotare la classe EsempioTextField di un metodo JTextField getTextField() che verra' usato dalla nostra classe ascoltratrice, invocandolo sull'istanza della finestra di cui avra' un riferimento. Nel nostro caso, il codice e' il seguente:

```
public class EsempioTextFieldListener implements ActionListener {

    // riferimento alla finestra
    private EsempioTextField frame;

    EsempioTextFieldListener(EsempioTextField frame) { //quando costruiamo
        l'ascoltatore quindi dobbiamo passargli un riferimento al frame in cui
        c'e' il componente presso cui registreremo questo ascoltatore
        this.frame = frame;
    }

    public void actionPerformed(ActionEvent a) {
        // Accede al componente "campoTesto" tramite

```

```

        // opportuno metodo
        JOptionPane.showMessageDialog(
            null, frame.getTextField().getText());
    }
}

-----
public class EsempioTextField extends JFrame {

    // Definizione di: centro, sud, campoTesto,
    // button, titolo (come sopra)
    private static final JTextField campoTesto = new JTextField(20);

    public EsempioTextArea() {
        // Creazione e impostazione finestra (come sopra)
        // Passiamo il riferimento alla finestra (this) al costruttore
        // del listener
        button.addActionListener(new EsempioTextFieldListener(this));
        this.pack();
        this.setVisible(true);
    }

    // restituisce riferimento al campo testuale. E' il metodo di cui parlavamo
    // prima.
    public JTextField getTextField() {
        return campoTesto;
    }
}

```

Un modo diverso di fare questo metodo 3 (ma sempre metodo 3 rimane) e' il seguente: invece di mettere i campi dati della finestra private e utilizzare dei metodi getter public da far usare ai listener, possiamo:

- Creare un package che conterra' tutte le classi relative alla nostra GUI. Questo in realta' lo facciamo naturalmente sempre
- Aggiungiamo un sottopackage per ogni finestra. Impostiamo come non qualificate le variabili della finestra. Oltre alla finestra, nel sottopackage ci saranno tutte le classi ascoltatrici relative a quella finestra. Essendo non qualificate, tutti i listener avranno accesso diretto alle variabili della finestra.
- Se ci sono ascoltatori che sono relativi a piu' finestre, metti tutto dentro un solo sottopackage. Quindi nel sottopackage ci saranno piu' finestre con tutti i loro listener.

Ovviamente questa soluzione diminuisce l'information hiding e aumenta l'accoppiamento, ma ne vale la pena quando i listener sono complessi e interagiscono con piu' finestre.

35.3 Limitare la proliferazione dei Listener:

Al crescere del numero dei componenti, cresce il numero di listener da implementare. Questo puo' diventare un problema non tanto di risorse (come invece dicono le slide) ma quanto di ordine, dato che se hai tanti listener poi il codice diventa illegibile. Allora possiamo usare un listener per piu' componenti, invece di uno per componente (come d'altronde abbiamo gia' visto e fatto), per limitare la proliferazione dei listener. Cio' vuol dire che dobbiamo creare dei listener che raggruppino piu' funzionalita' tra loro omogenee. Ok, in questo modo limitiamo la proliferazione dei listener, ma abbiamo un altro problema: come facciamo a riconoscere quale componente di volta in volta e' sorgente dell'evento se il listener ha piu' componenti associategli? Prima abbiamo usato il metodo getSource() e fatto il casting, ma questa procedura generalmente non si fa, poiche' e' sporca e puo' creare problemi. In generale, per risolvere questo problema si **usa il metodo setActionCommand(String c)**. Questo metodo permette di associare un comando, cioe' una *stringa identificativa*, ad un componente. In questo modo quando quest'ultimo genera un evento la stringa comando viene inserita nell'oggetto ActionEvent generato, che ricordiamo essere un oggetto che incapsula tutte le informazioni relative al cambio di stato del componente sorgente. Oltre al metodo getter, abbiamo ovviamente anche il metodo setter: String **getActionCommand()** della classe ActionEvent. Possiamo far usare questo metodo dal listener, invocandolo sull'istanza

dell'oggetto Evento che è stato generato per ottenere la stringa comando, ed identificare quale componente è stata a generare l'evento. In questo modo possiamo, tramite if-else o switch, far variare la gestione dell'evento da parte del listener in base alla stringa comando letta, ma non solo: dati due componenti diversi (es. una voce di un menu e un bottone), se questi devono essere gestiti nella stessa maniera, possono avere una stringa comando uguale in modo che vadano a finire nello stesso ramo if (es. voce di menu salva e bottone salva che devono fare la stessa cosa). Facciamo un esempio con il caso 3 di creazione dei listener, cioè classe esterna listener:

```

public class Listener implements ActionListener {

    // Costanti usate quando la stringa "comando" viene assegnata
    public static final String UPOPT = "up";
    public static final String DOWNOPT = "down";
    public static final String RANDOMOPT = "random";

    public void actionPerformed(ActionEvent e) {
        // legge il comando dall'evento
        String com = e.getActionCommand();
        // L'evento viene gestito in base al comando
        // (se non è una stringa valida, nessuna azione viene eseguita)
        if (com == UPOPT)
            upOpt();
        else if (com == DOWNOPT)
            downOpt();
        else if (com == RANDOMOPT)
            randomOpt();
    }

    // Metodi specifici invocati in base alla stringa "comando" ricevuta, nota che sono
    // privati perché li usa solo il Listener
    private void upOpt() { ... }
    private void randomOpt() { ... }
    private void downOpt() { ... }

}

```

E quindi, nel frame:

```

public class MyFrame extends JFrame {

    // ...
    JMenuItem upOpt = new JMenuItem("Up");
    JMenuItem downOpt = new JMenuItem("Down");
    JMenuItem randomOpt = new JMenuItem("Random");
    Listener ascoltatore = new Listener();

    public MyFrame() {
        // costruzione frame e menu ...

        // lo stesso listener viene associato alle diverse voci di menu
        upOpt.addActionListener(ascoltatore);
        // upOpt scrive nell'ActionEvent generato la stringa Listener.UPOPT
        upOpt.setActionCommand(Listener.UPOPT);

        downOpt.addActionListener(ascoltatore);
        // downOpt scrive nell'ActionEvent
        // generato la stringa Listener.DOWNOPT
        downOpt.setActionCommand(Listener.DOWNOPT);

        randomOpt.addActionListener(ascoltatore);
    }
}

```

```

// randomOpt scrive nell'ActionEvent
// generato la stringa Listener.RANDOMOPT
randomOpt.setActionCommand(Listener.RANDOMOPT);

// ...
}


```

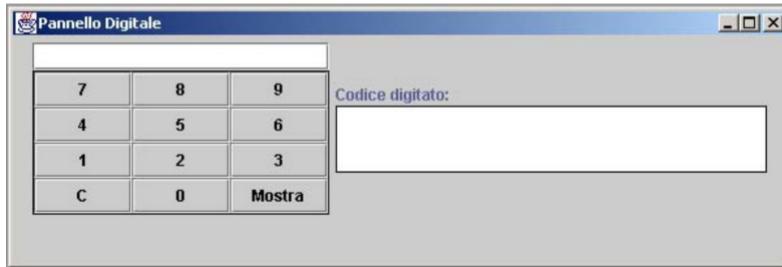
Notare come i controlli if-else nel Listener vengano fatti con "==" anche se si tratta di stringhe, e i confronti con stringhe vanno fatti con equals. In questo caso e' possibile usare "==" perche' com viene da ActionCommand, e ActionCommand e' nel Frame impostato con una delle costanti OPT. **Quando si e' sicuri di lavorare con costanti, il riferimento alla stringa che si ottiene punta alla costante, quindi ovviamente punta allo stesso oggetto della costante, e perciò "==" che verifica che com e la costante siano lo stesso oggetto funziona.** Questo pero' e' un caso particolare, perche' confrontiamo superficialmente una costante con una riferimento a stringa che punta alla stessa costante; se non avessimo usato una costante ma scritto in ActionCommand direttamente la stringa contenuta dentro la costante "==" non avrebbe funzionato, ma avremmo dovuto fare un **controllo di uguaglianza profonda con equals(), come facciamo sempre nel caso di confronti tra stringhe.** Quindi per poter utilizzare "==" dobbiamo essere sicuri di star usando le costanti nell'action command, non semplicemente di star inserendo una stringa uguale a quella contenuta dentro la costante. Nel dubbio, conviene sempre utilizzare equals() cosi uno va sul sicuro sempre. Quindi prendi come regola che se c'e' un confronto tra stringhe devi usare equals().

Ok questo era il modo di scrivere un programma con un solo listener che raccoglie piu' funzionalita'. Ovviamente questa tecnica per limitare la proliferazione dei listener puo' facilmente essere usata anche per gli altri 2 metodi. Quindi in pratica usiamo gli actionCommands per distinguere tra differenti sorgenti in una finestra, potendo così usare un solo listener.

36 Laboratorio 7 dell'11-11:

36.1 Esercizio 1:

Lo scopo di questa esercitazione è la realizzazione in Java di un'applicazione la cui interfaccia grafica è qui mostrata



La finestra principale è organizzata in due pannelli (`JPanel`) disposti secondo il layout *FlowLayout*. Il primo pannello (a sinistra) è costituito da due componenti collocati utilizzando le politiche del *BorderLayout*. Il componente posizionato a nord è costituito da un `JPasswordField` che mostra il testo introdotto sostituendo ogni carattere con un asterisco. Il componente centrale è costituito da un pannello, strutturato secondo il layout *GridLayout* con 4 righe e 3 colonne, che contiene i 12 pulsanti come in figura.

Il secondo pannello (a destra), i cui elementi sono disposti secondo *BorderLayout*, contiene due componenti: a nord una `JLabel` con il testo in figura, ed al centro una `JTextArea` (nota: il bordo intorno all'area di testo (o altro componente/pannello) può essere impostato usando il metodo `setBorder`. Ad esempio:

```
setBorder(BorderFactory.createLineBorder(Color.black))
```

 crea un bordo di colore nero intorno al componente/pannello oggetto d'invocazione.

L'utente interagisce con il programma premendo i pulsanti a sinistra. Ogni volta che un bottone corrispondente ad un numero viene premuto, il numero corrispondente viene inserito in coda nel campo password e mostrato come un asterisco. Non deve essere possibile modificare direttamente il testo né nel `JPasswordField` né nella `JTextArea` (nota: per fare ciò occorre impostare la proprietà *Editable* dei componenti al valore `false`).

Il bottone *Mostra* ha l'unico scopo di copiare la sequenza inserita, finora visualizzata solo come sequenza di asterischi, nell'area di testo a destra. Il bottone *C* serve a svuotare l'area di testo ed il campo password per inserire un nuovo codice segreto. Per evitare di cancellare erroneamente una codice inserito, dopo aver premuto il bottone *C* e prima dell'effettiva eliminazione, si può prevedere una finestra di conferma SI/NO. Se l'utente preme NO, l'operazione di cancellazione viene abortita.

Ecco la realizzazione:

```
package GuiApp;  
  
import java.awt.BorderLayout;  
import java.awt.Color;
```

La finestra di conferma può essere realizzata utilizzando il metodo statico `showConfirmDialog` della classe `JOptionPane` come segue:

- `JOptionPane.showConfirmDialog(null, "Cancellare il codice inserito?", "Pannello digitale", JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE)`

L'invocazione di questo metodo restituirà:

- la costante `JOptionPane.YES_OPTION`, se viene premuto SI;
- la costante `JOptionPane.NO_OPTION` se viene premuto NO.

```
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextArea;

public class MyFrame extends JFrame { //guarda schema gerarchico di progetto

    private static final String TITOLO = "Pannello Digitale";

    //Pannello di Sinistra
    private final JPanel pannelloDigitazione = new JPanel(new BorderLayout());
    final JPasswordField passwordNorth = new JPasswordField(); //nota che questo campo e'
        final perche' la variabile non puo' essere modificata. Questo non significa che
        l'istanza non puo' essere modificata. In altre parole, l'utente potra' modificare
        quello che c'e' scritto dentro il campo, ma non potro' mettere un nuovo
        JPasswordField in questa variabile
    private final KeyPad tastierino = new KeyPad(new MyActionListener(this)); //passiamo
        da qui il listener al KeyPad: questo e' fondamentale dato che il
            //listener deve avere un riferimento al
            MyFrame, e non possiamo darglielo
            //direttamente dal Keypad, ma dobbiamo
            passare per forza per di qua

    //Pannello di Destra
    private final JPanel pannelloDigitato = new JPanel(new BorderLayout());
    private final JLabel etichettaNord = new JLabel("Codice digitato:");
    final JTextArea casellaCentro = new JTextArea(3, 30); //3 righe e 30 colonne

    MyActionListener ascoltatore = new MyActionListener(this);

    public MyFrame() {
        super(TITOLO);

        //Pannello di Sinistra
        passwordNorth.setEditable(false); //non si puo' modificare il testo all'interno
        pannelloDigitazione.add(passwordNorth, BorderLayout.NORTH);
```

```

pannelloDigitazione.add(tastierino, BorderLayout.CENTER);

//Pannello di Destra
casellaCentro.setBorder(BorderFactory.createLineBorder(Color.black)); //Crea un
    bordo di colore nero intorno all'oggetto di invocazione
casellaCentro.setEditable(false); //non si puo' modificare il testo all'interno
pannelloDigitato.add(etichettaNord, BorderLayout.NORTH);
pannelloDigitato.add(casellaCentro, BorderLayout.CENTER);

this.getContentPane().setLayout(new FlowLayout());
this.getContentPane().add(pannelloDigitazione);
this.getContentPane().add(pannelloDigitato);

this.setDefaultCloseOperation(EXIT_ON_CLOSE); //invece di JFrame.EXIT_ON_CLOSE
funziona??
this.pack();

this.setVisible(true);
}

}

-----
package GuiApp;

import java.awt.Color;
import javax.swing.JButton;
import javax.swing.JPanel;
import java.awt.GridLayout;
import javax.swing.BorderFactory;

public class KeyPad extends JPanel { //Il tastierino e' una classe a se'
    //nota che deve estendere JPanel

    static final String CLEAR_BUTTON = "C";
    static final String SHOW_BUTTON = "mostra";

    private final JButton[] buttoniNumerici = new JButton[10];
    private final JButton bottoneClear = new JButton("C");
    private final JButton bottoneMostra = new JButton("mostra");

    public KeyPad(MyActionListener ascoltatore) {

        this.setLayout(new GridLayout(4,3));

        for (int i = 0; i < buttoniNumerici.length; i++) {
            buttoniNumerici[i] = new JButton(i + ""); //modo rapido per trasformare
                //un intero i in una stringa
            buttoniNumerici[i].setActionCommand(i + "");
            buttoniNumerici[i].addActionListener(ascoltatore);
            /*int col = i % 3;           //questa e' una cosa che il prof

```

```

        int row = i % 3;           //ha provato a fare ma non funziona
        this.add(buttoniNumerici[i], row, col);/* //ce lo dira' a lezione

    }

    bottoneClear.setActionCommand(CLEAR_BUTTON);
    bottoneClear.addActionListener(ascoltatore);
    bottoneMostra.setActionCommand(SHOW_BUTTON);
    bottoneMostra.addActionListener(ascoltatore);

    this.add(buttoniNumerici[7]);
    this.add(buttoniNumerici[8]);
    this.add(buttoniNumerici[9]);
    this.add(buttoniNumerici[4]);
    this.add(buttoniNumerici[5]);
    this.add(buttoniNumerici[6]);
    this.add(buttoniNumerici[1]);
    this.add(buttoniNumerici[2]);
    this.add(buttoniNumerici[3]);
    this.add(bottoneClear);
    this.add(buttoniNumerici[0]);
    this.add(bottoneMostra);

    this.setBorder(BorderFactory.createLineBorder(Color.black));
}

}

-----
package GuiApp;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JOptionPane;
import javax.swing.JPasswordField;

public class MyActionListener implements ActionListener {

    private final MyFrame riferimentoFrame;

    public MyActionListener(MyFrame riferimentoFrame) {
        this.riferimentoFrame = riferimentoFrame;
    }

    @Override
    public void actionPerformed(ActionEvent e) {

        if (e.getActionCommand().equals(KeyPad.CLEAR_BUTTON)) { //usiamo l>equals per
            //confrontare le stringhe, ma,
            //trattandosi di costanti, si
            //suppone che quando si
            //inserisce l'action command si
            //vada a inserire la costante
            //e non la stringa in
            //quest'ultima contenuta
            //in questo modo
            //l'actioncommand punterà
            //allo stesso oggetto
    }
}

```

```

                //della costante, quindi
                //possiamo senza problemi
                //usare anche
                //==" , in questo caso.
                //Comunque per non sbagliare,
                //uno va
                //sul sicuro con equals OGNI
                //VOLTA CHE SI TRATTA DI
                //STRINGHE

        int YES_NO = JOptionPane.showConfirmDialog(null, "Cancellare il codice
                inserito?", "Richiesta Conferma",
                JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE); //l'ultimo
                //parametro e' per l'icona

        if (YES_NO == JOptionPane.YES_OPTION) {
            riferimentoFrame.passwordNorth.setText(""); //metodo ereditato da
            JTextField
            riferimentoFrame.casellaCentro.setText("");
        } else if (YES_NO == JOptionPane.NO_OPTION) {
            //non fare nulla
        }

    }

    else if (e.getActionCommand().equals(KeyPad.SHOW_BUTTON)) {
        String password =
            String.valueOf(riferimentoFrame.passwordNorth.getPassword()); //getText
            e' deprecated quindi NON VA USATO
        // altro modo di farlo: String precedente = new
        // String(riferimentoFrame.passwordNorth.getPassword());
        riferimentoFrame.casellaCentro.setText(password);
    }

    else { //si tratta per forza di un bottone corrispondente a un numero
        String precedente =
            String.valueOf(riferimentoFrame.passwordNorth.getPassword());
        riferimentoFrame.passwordNorth.setText(precedente + e.getActionCommand());
    }

}

-----
package GuiApp;

public class Main {

    public static void main(String[] args) {
        new MyFrame();
    }

}

```

36.2 Esercizio 2:

Si vuole realizzare la seguente applicazione.



La finestra dovrà essere composta da tre campi testuali (Matricola, IP Address, Porta) e da quattro pulsanti (Connect, Disconnect, Start, Stop) disposti secondo l'immagine riportata

Si implementi il seguente protocollo:

- Alla pressione del pulsante "Connect", l'applicazione deve inviare una richiesta di connessione utilizzando indirizzo IP e porta indicate nei campi testuali "IP Address" e "Porta". La richiesta di connessione (ai fini dell'esercizio odierno) consiste nello scrivere sul log (cf. logging in Java) la stringa `connect <IP Address>:<Porta>`.
- Alla pressione del pulsante "Start", dopo la connessione, l'applicazione deve mandare il comando `start`. Anche in questo caso, ai fini dell'esercizio odierno, mandare il comando significa scrivere sul log la stringa `start`.
- Alla pressione del pulsante "Stop", l'applicazione deve mandare il comando `stop`. Di nuovo, ai fini dell'esercizio odierno, mandare il comando significa scrivere sul log la stringa `stop`.
- Alla pressione del pulsante "Disconnect" il client invia il comando `disconnect`. Di nuovo, ai fini dell'esercizio odierno, mandare il comando significa scrivere sul log la stringa `disconnect`.

Gestire correttamente la possibilità di premere i pulsanti, in particolare:

- all'avvio solo il pulsante "Connect" è abilitato
- una volta avviata una connessione, non deve poter essere premuto il pulsante "Connect"
- "Disconnect" può essere premuto solo se una connessione è avviata
- "Stop" può essere premuto solo se il client è connesso
- "Start" può essere premuto solo se il client è connesso ma non è stato fermato.

Un indirizzo IP è un codice univoco assegnato a una macchina all'interno di una rete e una porta è un codice univoco assegnato a un servizio all'interno di una macchina. Quindi facendo un paragone con le mappe geografiche l'IP è l'indirizzo e la porta è l'interno del palazzo. Nota che, non avendo ancora visto le socket, non sappiamo ancora fare una connessione di rete, quindi questa funzionalità è solo una scrittura del log. Poi puo' essere che rivediamo questi esercizi quando tratteremo i socket. Questa è la realizzazione:

```
package AddressIPGUIApp;

import java.awt.BorderLayout;
import java.awt.GridLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class MyFrame extends JFrame { //guarda schema gerarchico di progetto

    final JPanel centerPanel = new JPanel(new GridLayout(2,2));
    final JPanel southPanel = new JPanel(); // Flow by default

    final JPanel portaPanel = new JPanel(new BorderLayout());
    final JPanel matricolaPanel = new JPanel(new BorderLayout());
    final JPanel IpPanel = new JPanel(new BorderLayout());

    final JButton connectButton = new JButton("Connect");
    final JButton disconnectButton = new JButton("Disconnect");
    final JButton startButton = new JButton("Start");
    final JButton stopButton = new JButton("Stop");

    final JLabel northPortaLabel = new JLabel ("Porta");
    final JTextField centerPortaTextField = new JTextField(30);
    final JLabel northMatricolaLabel = new JLabel ("Matricola");
    final JTextField centerMatricolaTextField = new JTextField(30);
    final JLabel northIpLabel = new JLabel ("IP Address");
```

```

final JTextField centerIpTextField = new JTextField(30);

public MyFrame() {

    disconnectButton.setEnabled(false); //inizialmente funziona solo
    startButton.setEnabled(false);     //il tasto connect
    stopButton.setEnabled(false);

    matricolaPanel.add(northMatricolaLabel, BorderLayout.NORTH);
    matricolaPanel.add(centerMatricolaTextField, BorderLayout.CENTER);

    portaPanel.add(northPortaLabel, BorderLayout.NORTH);
    portaPanel.add(centerPortaTextField, BorderLayout.CENTER);

    IpPanel.add(northIpLabel, BorderLayout.NORTH);
    IpPanel.add(centerIpTextField, BorderLayout.CENTER);

    centerPanel.add(matricolaPanel);
    centerPanel.add(IpPanel);
    centerPanel.add(portaPanel);

    MyActionListener ascoltatore = new MyActionListener(this);

    connectButton.addActionListener(ascoltatore);
    connectButton.setActionCommand(MyActionListener.CONNECT);
    disconnectButton.addActionListener(ascoltatore);
    disconnectButton.setActionCommand(MyActionListener.DISCONNECT);
    stopButton.addActionListener(ascoltatore);
    stopButton.setActionCommand(MyActionListener.STOP);
    startButton.addActionListener(ascoltatore);
    startButton.setActionCommand(MyActionListener.START);

    southPanel.add(connectButton);
    southPanel.add(disconnectButton);
    southPanel.add(startButton);
    southPanel.add(stopButton);

    this.getContentPane().add(centerPanel, BorderLayout.CENTER);
    this.getContentPane().add(southPanel, BorderLayout.SOUTH);

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.pack();
    this.setVisible(true);
}

-----


package AddressIPGUIApp;

import java.util.logging.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyActionListener implements ActionListener {

```

```

private MyFrame riferimentoFinestra;

static final String CONNECT = "connect"; //nell'esercizio di prima abbiamo messo le
costanti nel Frame
static final String DISCONNECT = "disconnect"; //(precisamente in un Panel del
Frame), ora le mettiamo
static final String START = "start"; //nel Listener. Non fa alcuna differenza,
scegli il modo che
static final String STOP = "stop"; //preferisci tra i due.

private final Logger logger = Logger.getLogger("AddressIPGUIApp.MyLogger");
//should it be static and final?

public MyActionListener(MyFrame riferimentoFinestra) {
    this.riferimentoFinestra = riferimentoFinestra;
}

@Override
public void actionPerformed(ActionEvent e) {

    if (e.getActionCommand() == CONNECT) {

        riferimentoFinestra.disconnectButton.setEnabled(true); //se connettiamo gli
        altri
        riferimentoFinestra.startButton.setEnabled(true); //tasti si attivano
        riferimentoFinestra.stopButton.setEnabled(true);

        riferimentoFinestra.connectButton.setEnabled(false); //se premo connect non
        posso riprenderlo

        String IpAddress = riferimentoFinestra.centerIpTextField.getText();
        String Porta = riferimentoFinestra.centerPortaTextField.getText();
        logger.log(Level.INFO, "connect <" + IpAddress + ">:<" + Porta + ">");

    }

    else if (e.getActionCommand() == DISCONNECT) {

        riferimentoFinestra.disconnectButton.setEnabled(false); //se disconnettiamo
        gli altri
        riferimentoFinestra.startButton.setEnabled(false); //tasti tornano disattivati
        riferimentoFinestra.stopButton.setEnabled(false);

        riferimentoFinestra.connectButton.setEnabled(true); //se premo connect non
        posso riprenderlo

        logger.log(Level.INFO, "disconnect");

    }

    else if (e.getActionCommand() == START) {

        logger.log(Level.INFO, "start");

    }
}

```

```

        else if (e.getActionCommand() == STOP) {

            riferimentoFinestra.startButton.setEnabled(false);

            logger.log(Level.INFO, "stop");

        }

    }

}

-----
package AddressIPGUIApp;

public class Main {

    public static void main(String[] args) {
        MyFrame finestraUno = new MyFrame();
    }

}

```

Vediamo che per creare il logger non si fa new Logger ma si usa il metodo statico getLogger() della classe Logger, passandogli in input il nome del Logger che vogliamo creare. Questa e' una modalita' di istanziazione che viene fatta spesso nel design pattern factory, che è un pattern di una classe. Se una classe ha il design pattern factory è una classe che crea altre istanze. Nel design pattern factory per le classi il costruttore è in genere inibito (es. messo a private), per questo motivo non può essere usato e dobbiamo usare i metodi factory, come in questo caso, in cui usiamo getLogger(). Quindi Logger è una classe factory. Nota inoltre che non abbiamo configurato noi il logging, quindi stiamo usando i parametri standard di default definiti dal file logging.properties. Per questo motivo, il log ci verra' mostrato solo su console.

36.3 Shortcut da Tastiera:

Gli shortcut da tastiera per i bottoni si fanno con:

```
bottone.setMnemonic(KeyEvent.VK_CARATTERE);
```

Quindi potremo cliccare il pulsante con Alt + CARATTERE. Gli shortcut da tastiera per le JTextField invece sono:

```
JLabel etichettaPort = new JLabel("Port");
JTextField port = new JTextField();

etichettaPort.setLabelFor(this.port))

pannello.add(etichettaPort);
pannello.add(port);

etichettaPort.setDisplayedMnemonic(KeyEvent.VK_Carattere);
```

37 L'istruzione Switch:

Permette la verifica del valore di un'espressione, che viene confrontata con dei valori appartenenti a dei casi. Se l'espressione ha un valore uguale a quello di uno dei casi previsti, viene eseguito il

corrispondente codice. Esempio:

```
Switch(e.getActionCommand()) {  
  
    case ClientListener.CONNECT:  
        //fai qualcosa  
        break;  
    case ClientListener.DISCONNECT:  
        //fai qualcosa  
        break;  
    case ClientListener.IMAGE1:  
    case ClientListener.IMAGE2:  
    case ClientListener.IMAGE3:  
    case ClientListener.IMAGE4:  
    case ClientListener.IMAGE5:  
        //fai qualcosa che vale per tutti e 5 gli Image Case  
        break;  
}
```

Part VIII

Logging:

Il logging e' la registrazione delle informazioni sull'esecuzione del codice. Il log e' il file o insieme di file su cui tali registrazioni sono memorizzate. Memorizzare cio' che e' successo nel programma e' utile ad esempio per capire perche' si e' verificato un certo errore, o per accountability (responsabilita'), cioe' per capire chi ha fatto cosa. Il logging puo' essere sostituito dal piu' tradizionale e semplice utilizzo di stampe all'interno del programma. Questo pero', anche se e' un metodo veloce, ha importanti limitazioni:

- Le stampe sono tutte mandate sullo standard output, cioe' sulla console. Se io voglio memorizzare queste informazioni in un file diventa quindi scomodo.
- e' necessario ogni volta modificare il codice per commentare-decommentare le stampe di debug.

Capiamo quindi che e' preferibile avere strumenti appositi per il logging: i sistemi di logging. Un sistema di logging e' una libreria di classi che include nel codice delle stampe di debug che descrivono il funzionamento del programma, ma solo nelle sessioni di correzione, poi queste stampe vengono eliminate in versioni di produzione. Cio' che differenzia queste stampe del sistema di logging da quelle che facciamo noi tradizionalmente e':

- Scegliamo piu' facilmente quali livelli di gravita' (v. dopo) delle informazioni dobbiamo visualizzare e quali non ci interessano, senza dover modificare a mano ogni volta tutte le stampe.
- Consente di registrare su dispositivi e con formati diversi queste informazioni, e non solo sulla console.

Il sistema di logging standard di Java e' la libreria `java.util.logging`³⁶.

37.1 Concetti Principali del Logging:

37.1.1 Messaggio di Logging (log records):

Si tratta di una stringa registrata durante l'esecuzione di un metodo. Questa stringa contiene una serie di informazioni su cosa e come il metodo ha fatto (es. timestamp, livello di gravita' ...).

37.1.2 Logger:

Il logger e' la componente che si occupa della creazione dei messaggi di logging. Possiamo avere piu' logger per la stessa applicazione, per diversi motivi. Ogni logger ha un nome ed e' organizzato in una gerarchia di logger, con un logger principale detto Root Logger.

37.1.3 Hadler (o Appender):

L'Handler e' la componente che si occupa di registrare il messaggio di logging prodotto da un logger all'interno di uno specifico dispositivo. Se nel file di configurazione vogliamo i messaggi di logging su console e file, verranno inizializzati Handler per console e file; se vogliamo messaggi di logging solo su console allora verranno inizializzati Handler solo su console ecc.. Noi diciamo solo quali Handler vogliamo, poi la libreria si occupa di inizializzarli.

37.1.4 Livello di logging (filter):

e' il livello di gravita' di cui parlavamo prima. Il livello di logging descrive la gravita' di un messaggio. I tipici livelli di logging in Java, messi in ordine decrescente di gravita', sono:

- SEVERE: problemi molto gravi che possono portare alla chiusura del programma.

³⁶poi abbiamo la diffusa libreria `org.apache.log4j`. Ogni sistema di logging, come queste due librerie, ha dei metodi a se' stanti: non c'e' un'interfaccia standard sviluppata da Java su cui tutte si basano. Passare da una libreria all'altra non e' traumatico comunque, avendo tutte librerie simili, seppur non la stessa. In realtà oggigiorno e' spesso utilizzata l'interfaccia astratta `SLF4J`, che e' un'interfaccia standard per le librerie di logging, semplicemente non e' stata sviluppata ufficialmente da Java

- WARNING: avvertimenti. Non terminano il programma ma sono potenzialmente gravi.
- INFO: rilascia messaggi sul funzionamento del programma, quindi interessa anche all'utente finale.
- FINE: livello di debug non dettagliato. Questo livello quindi rilascia messaggi di tracing.
- FINER: livello di debug più dettagliato.
- FINEST: livello di debug dettagliato al massimo. Rilascia messaggi di tracing dettagliati.

A ciascun logger si associa un livello, così che esso ci farà visualizzare solo i messaggi da quel livello in su (così non dobbiamo ogni volta commentare-decommentare le stampe). Il livello standard - cioè quando il SW è presso il cliente - è INFO, ma è modificabile attraverso un file di configurazione. Il livello di logging scelto ha un forte impatto sulle performance, quindi bisogna scegliere il giusto livello di logging (es. se sto facendo debugging ha senso mettere livello FINEST, senno lascia a INFO).

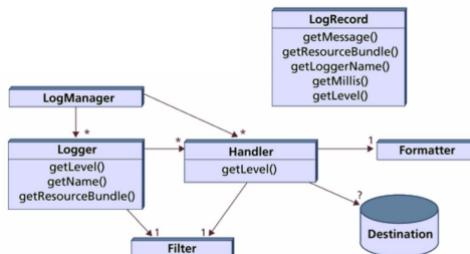
37.2 Il Sistema di Logging Standard di Java: `java.util.logging`:

37.2.1 La classe LogManager:

La classe LogManager è il Logger Root, quindi si occupa della gestione di tutti gli altri logger. Definisce due metodi:

- `getLogManager()` che restituisce il Manager
- `addLogger()` che aggiunge un logger
- `getLogger()` restituisce un logger

Per ogni componente da monitorare aggiungiamo un logger, e per ognuno di questi logger specifichiamo gli handler (cioè dove quel logger deve essere memorizzato) e un determinato livello di filtro, cioè il livello di gravità'.



37.3 Configurazione del Logging:

I parametri di configurazione (es. il numero e il tipo di Handlers), usati dal LogManager, vengono letti da un file `logging.properties`. Quello di default si trova nella directory `lib` (che a sua volta è nella directory `Java` del JRE (Java Runtime Environment)). Possiamo modificare il file di properties per modificare questi parametri. La configurazione di default prevede due Handler globali, uno su console e uno su file nella home directory utente (anche se questo secondo è commentato, quindi disattivato, di default). Il livello di logging di default è INFO. Questi parametri possono essere modificati anche a runtime, attraverso opportuni metodi, in questo modo possiamo a runtime:

- creare nuovi Handlers
- creare nuovi Loggers
- cambiare il livello di gravità' dei Loggers (con il metodo `setLevel` di `LogManager` e `Logger`).

Si associa un file `logging.properties` ad ogni programma.

37.4 Vediamo un esempio di Programma che fa Logging:

```
package it.uniroma1.diag;

import java.util.logging.*;

public class Loggata {

    // ottiene un logger
    private static Logger logger = Logger.getLogger("it.uniroma1.diag.Loggata"); // per
        il nome del logger si usa spesso questa notazione di tipo package. Questo nome e'
        importante perche' adesso tutti i messaggi che verranno inviati con questo logger
        verranno etichettati con questa etichetta. Quindi posso filtrare i messaggi in
        base al loro nome.

    private static void metodoEsempio() {
        System.out.println("... sto eseguendo un'operazione complessa ...");
        // sto eseguendo un'operazione che puo' generare eccezioni
    }

    public static void main(String args[]) {

        try {
            Loggata.metodoEsempio();
        } catch (Exception ex) {
            // Log the exception
            logger.log(Level.WARNING, "problemi nel metodo", ex); //log e' il metodo che
                mi permette di creare un nuovo messaggio, il primo parametro e' il livello
                (Level e' una classe in java.util.logging che contiene una serie di
                costanti). Si puo' personalizzare il color codes del logger per utilizzare
                colori diversi per diversi livelli di gravita'.
        }

        // Log un messaggio di tracing INFO
        // provare ad usare Level.FINE e vedere cosa succede
        logger.log(Level.INFO, "fatto");
    }
}
```

Supponiamo che metodoEsempio() non generi nessun errore. Allora otteniamo:



The screenshot shows a Java application running in an IDE. The console tab displays the following output:

```
<terminated> Loggata [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java (Dec 10,
... sto eseguendo un'operazione complessa ...
Dec 10, 2016 11:06:29 AM it.uniroma1.diag.Loggata main
INFO: fatto
```

37.5 Un altro esempio:

Stavolta l'Handler sara' su file, quindi da console non vedremo nulla. Nel file di log troveremo questo (in XML): Vediamo che noi abbiamo, per ogni log, semplicemente passato un messaggio e un livello di gravita', ma le informazioni che sono state registrate sono molte di piu'. Precisamente, per ogni messaggio c'e' uno specifico tag di tipo record. Avendo due messaggi log abbiamo due tag di tipo record. Per ogni record abbiamo `jdate`, `jmillis`, per il timestamp; `jsequence`, che indica l'ordinale all'interno del logger, quindi nel nostro caso per il primo record abbiamo ovviamente 0 e per il secondo 1; poi abbiamo `jlogger`, che ci mostra il messaggio che abbiamo passato quando abbiamo costruito il logger, quindi e' il nome del logger; poi abbiamo `jlevel`, che indica il livello di gravita' del messaggio; `jclass`, che ci mostra la classe dove e' stato generato il messaggio; `jmethod`, che ci mostra il metodo in cui e' stato generato il messaggio all'interno della classe; `jthread`, che ci dice qual e' il numero di thread dell'applicativo (vedremo dopo i thread); e infine `jmessage`, che ci mostra il vero e proprio messaggio. Tutto questo per ogni `jrecord`, quindi per ogni log, cioe' per ogni messaggio.

The screenshot shows the Eclipse IDE interface. At the top is a code editor window containing Java code for a class named Loggata_2. The code includes imports for java.io.IOException and java.util.logging.Level, and defines a static logger. It contains a main method that prints a message to the console, sets the logger level to ALL, adds a FileHandler to the logger, and logs FINE and WARNING levels. A try-catch block is present to catch exceptions and log them at the WARNING level. The code ends with a final FINE log message.

Below the code editor is a 'Console' tab showing the output of the program's execution:

```
<terminated> Loggata_2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java (Dr
... sto eseguendo un'operazione complessa ...


```

At the bottom is a 'Problems' tab and a 'Declaration' tab. The main workspace area displays the generated XML log file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
<date>2016-12-10T11:24:44</date>
<millis>1481365484378</millis>
<sequence>0</sequence>
<logger>it.uniroma1.diag.Loggata_2</logger>
<level>FINE</level>
<class>it.uniroma1.diag.Loggata_2</class>
<method>main</method>
<thread>1</thread>
<message>tutto bene</message>
</record>
<record>
<date>2016-12-10T11:24:44</date>
<millis>1481365484404</millis>
<sequence>1</sequence>
<logger>it.uniroma1.diag.Loggata_2</logger>
<level>FINE</level>
<class>it.uniroma1.diag.Loggata_2</class>
<method>main</method>
<thread>1</thread>
<message>fatto</message>
</record>
</log>
```

37.6 Personalizzare il logging:

Solitamente si crea un logger per ogni classe di cui interessa il logging, ma puoi fare come vuoi. Se la classe è fatta di tutti metodi statici si può fare un logger static, così non devi istanziare per forza, ma sennò è buona norma non averlo static. Scriviamo un primo programma che, non avendo modificato le properties, avrà i parametri di logging di default: Non vediamo il messaggio

The screenshot shows the Eclipse IDE interface. The code editor window contains a Main class with a usaLogging() method that logs INFO, WARNING, and FINE messages. The main() method creates an instance of Main and calls usaLogging(). The output window shows the log messages:

```
run:
Nov 15, 2021 7:36:59 AM Main usaLogging
INFO: Prova Info
Nov 15, 2021 7:37:00 AM Main usaLogging
WARNING: Avvertenza
BUILD SUCCESSFUL (total time: 0 seconds)
```

di debug perché il file di configurazione logging.properties ha sia come logging level globale sia come logging level su console INFO. Il logging globale .level = INFO è quello utilizzato se non

si specifica altrimenti; sui singoli handler mettiamo il livello noi, e anche lì è a info. Allora su NetBeans andiamo su source packages, new, e cerchiamo file properties. Creiamo un nuovo file logging.properties e per ora copiamoci e incolliamoci quello di default. Quello di default è:

```
#####
# Default Logging Configuration File
#
# You can use a different file by specifying a filename
# with the java.util.logging.config.file system property.
# For example java -Djava.util.logging.config.file=myfile
#####

#####
# Global properties
#####

# "handlers" specifies a comma separated list of log Handler
# classes. These handlers will be installed during VM startup.
# Note that these classes must be on the system classpath.
# By default we only configure a ConsoleHandler, which will only
# show messages at the INFO and above levels.
handlers= java.util.logging.ConsoleHandler

# To also add the FileHandler, use the following line instead.
#handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# Default global logging level.
# This specifies which kinds of events are logged across
# all loggers. For any given facility this global level
# can be overridden by a facility specific level
# Note that the ConsoleHandler also has a separate level
# setting to limit messages printed to the console.
.level= INFO                                //livello di logging globale (se no

#####

# Handler specific properties.
# Describes specific configuration info for Handlers.
#####

# default file output is in user's home directory.
# java.util.logging.FileHandler.pattern = %h/java%u.log      //vediamo che l'handler da file è com
# java.util.logging.FileHandler.limit = 50000
# java.util.logging.FileHandler.count = 1
# Default number of locks FileHandler can obtain synchronously.
# This specifies maximum number of attempts to obtain lock file by FileHandler
# implemented by incrementing the unique field %u as per FileHandler API documentation.
# java.util.logging.FileHandler.maxLocks = 100
# java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# Example to customize the SimpleFormatter output format
# to print one-line log message like this:
#      <level>: <log message> [<date/time>]
```

```

#
# java.util.logging.SimpleFormatter.format=%4$s: %5$s [%1$tc]%n
#####
# Facility specific properties.
# Provides extra control for each logger.
#####

# For example, set the com.xyz.foo logger to only log SEVERE
# messages:
com.xyz.foo.level = SEVERE

```

Ma non ho ancora specificato che voglio usare questo file di properties per il logging. Per fare ciò bisogna andare nell'opzione properties dell'intero progetto (icona del caffé), poi run, e come VM Option metti:

```
-Djava.util.logging.config.file = src/logging.properties
```

Queste VM Option sono opzioni passate alle Java Virtual Machine. In particolare -D indica che stiamo definendo una variabile d'ambiente, aggiuntiva rispetto a quelle dell'OS. Una variabile d'ambiente è una variabile definita a livello di Sistema Operativo, e sono usate per guidare quest'ultimo. La variabile d'ambiente classica è path, che dà una serie di percorsi in cui cercare gli eseguibili. Su Windows le variabili d'ambiente si trovano aprendo il prompt dei comandi e scrivendo set. Ogni programma può definire delle proprie variabili d'ambiente, in particolare noi lo facciamo con -D, definendo la variabile d'ambiente java.util.logging.config.file, e assegnandogli il valore ./logging.properties. Le variabili definite con -D sono attive solo per l'esecuzione del programma.

Adesso giochiamo un po' con questo file properties. Andiamo a decommentare l'handler da file, quindi

```

#####
# Default Logging Configuration File
#
# You can use a different file by specifying a filename
# with the java.util.logging.config.file system property.
# For example java -Djava.util.logging.config.file=myfile
#####

#####
# Global properties
#####

# "handlers" specifies a comma separated list of log Handler
# classes. These handlers will be installed during VM startup.
# Note that these classes must be on the system classpath.
# By default we only configure a ConsoleHandler, which will only
# show messages at the INFO and above levels.
handlers= java.util.logging.ConsoleHandler

# To also add the FileHandler, use the following line instead.
#handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# Default global logging level.
# This specifies which kinds of events are logged across
# all loggers. For any given facility this global level
# can be overridden by a facility specific level
# Note that the ConsoleHandler also has a separate level
# setting to limit messages printed to the console.

```

```

.level= INFO

#####
# Handler specific properties.
# Describes specific configuration info for Handlers.
#####

# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = %h/java%u.log           //decommentiamo l'handler da file. Ver
java.util.logging.FileHandler.limit = 50000                      //in formato XML, che abbiamo già
java.util.logging.FileHandler.count = 1
# Default number of locks FileHandler can obtain synchronously.
# This specifies maximum number of attempts to obtain lock file by FileHandler
# implemented by incrementing the unique field %u as per FileHandler API documentation.
java.util.logging.FileHandler.maxLocks = 100
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# Example to customize the SimpleFormatter output format
# to print one-line log message like this:
#      <level>: <log message> [<date/time>]
#
# java.util.logging.SimpleFormatter.format=%4$s: %5$s [%1$tc]%n

#####
# Facility specific properties.
# Provides extra control for each logger.
#####

# For example, set the com.xyz.foo logger to only log SEVERE
# messages:
com.xyz.foo.level = SEVERE

```

Vediamo che facendo il logging da file, ogni volta viene sovrascritto il file e rifatto. Possiamo fare in modo che ogni nuovo log venga aggiunto in coda al file. L'opzione:

```
java.util.logging.FileHandler.append
```

c'è scritto sulla Java Doc che "specifies whether the FileHandler should append onto any existing files (defaults to false)"; cioè di default si crea un nuovo log ogni volta. Mettendolo a true, o meglio aggiungendo proprio questa riga al nostro file (dato che non c'è), invece andiamo a realizzare quello che vogliamo.

```

#####
# Default Logging Configuration File
#
# You can use a different file by specifying a filename
# with the java.util.logging.config.file system property.
# For example java -Djava.util.logging.config.file=myfile
#####

#####
# Global properties
#####
```

```

# "handlers" specifies a comma separated list of log Handler
# classes. These handlers will be installed during VM startup.
# Note that these classes must be on the system classpath.
# By default we only configure a ConsoleHandler, which will only
# show messages at the INFO and above levels.
handlers= java.util.logging.ConsoleHandler

# To also add the FileHandler, use the following line instead.
#handlers= java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# Default global logging level.
# This specifies which kinds of events are logged across
# all loggers. For any given facility this global level
# can be overridden by a facility specific level
# Note that the ConsoleHandler also has a separate level
# setting to limit messages printed to the console.
.level= INFO

#####
# Handler specific properties.
# Describes specific configuration info for Handlers.
#####

# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
# Default number of locks FileHandler can obtain synchronously.
# This specifies maximum number of attempts to obtain lock file by FileHandler
# implemented by incrementing the unique field %u as per FileHandler API documentation.
java.util.logging.FileHandler.maxLocks = 100
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
java.util.logging.FileHandler.append = True //aggiungi

# Limit the message that are printed on the console to INFO and above.
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# Example to customize the SimpleFormatter output format
# to print one-line log message like this:
#      <level>: <log message> [<date/time>]
#
# java.util.logging.SimpleFormatter.format=%4$s: %5$s [%1$tc]%n

#####
# Facility specific properties.
# Provides extra control for each logger.
#####

# For example, set the com.xyz.foo logger to only log SEVERE
# messages:
com.xyz.foo.level = SEVERE

```

Ora, per ogni esecuzione del programma, viene creata una nuova sezione XML nel file.

37.7 Impostare un livello di gravità per un logger:

Sappiamo che esiste da codice sorgente il metodo `setLevel()` di `Logger` da invocare su un logger per impostargli un livello di logging. Questa cosa si può fare anche nel file `logging.properties`. Se notiamo l'ultima riga del file, troviamo:

```
# For example, set the com.xyz.foo logger to only log SEVERE
# messages:
com.xyz.foo.level = SEVERE
```

Questo è solo un esempio, se andiamo a mettere al posto di questo nome il nome etichetta del nostro vero logger:

```
it.diag.uniroma1.Main.level = SEVERE
```

Vediamo che andando a eseguire gli handler non scrivono nulla, perché abbiamo messo per questo logger il livello a `SEVERE` e noi abbiamo tutti messaggi di livello inferiore a questo. Questa cosa è consigliata farla proprio qui, sul file `logging.properties`, ma se vogliamo possiamo farlo anche da codice con `setLevel()`, come già detto.

Part IX

Input/Output:

Quello che abbiamo usato fino ad ora per gestire l'input/output è solo la punta dell'iceber, adesso invece lo vediamo in dettaglio. L'input è l'attività di lettura dal mondo esterno, mentre l'Output è la scrittura nel mondo esterno. Il mondo esterno rispetto all'applicazione non è per forza un altro calcolatore, ma riguarda anche altri file e altre applicazioni nello stesso calcolatore. Che vuol dire comunicare con altre app nello stesso pc? I sistemi operativi moderni isolano le applicazioni, tramite il meccanismo di memoria virtuale. In questo modo l'app ha la sensazione di essere l'unica che gira sul pc. Ci sono poi dei meccanismi a code, o di memoria condivisa (o di altro tipo) che servono a fare comunicare queste app. In questo corso noi vediamo solo la comunicazione con altre applicazioni tramite meccanismi di rete. Java per gestire uniformemente tutti i differenti tipi di comunicazione usa gli stream, cioè dei flussi di comunicazione input/output. In questo modo, così come abbiamo usato println per scrivere da console possiamo usare println anche su file, perciò il meccanismo è sempre lo stesso e cambia soltanto lo specifico stream su cui la println verrà eseguita. Possiamo vedere lo stream come un canale (tubo) attraverso cui passano delle informazioni. Gli stream sono implementati con classi contenute nel package.java.io e sono monodirezionali, cioè abbiamo stream di input e stream di output. System.in è uno stream di input, System.out è uno stream di output (in realtà è un print writer che è qualcosa di leggermente differente). I dispositivi esterni con cui comunichiamo possono essere delle sorgenti (es. tastiera), delle destinazioni (es. il video), oppure entrambi (es. file che è sia sorgente che destinazione, anche se un file o lo apro in scrittura o lo apro in lettura, non è quindi sorgente e destinazione contemporaneamente per lo stesso stream, ma ne devo avere due, essendo monodirezionali). Ci sono diverse categorizzazioni degli stream oltre a quella di input e output: altro modo di categorizzarli e tra stream di byte e stream di caratteri. Uno stream di byte ha i numeri rappresentati con byte che rappresentano i numeri (quindi per un intero ad esempio ho 4 byte), ma se rappresento un numero in uno stream di caratteri ho le sue cifre. (es. 32 in stream di caratteri è "3" e "2"). Gli stream di caratteri permettono di non occuparsi della conversione, perché se ne occupa direttamente Java (anche perché sennò erano solo un tipo particolare di stream di byte, ed era inutile differenziare gli stream rispetto a questi). Java adotta la codifica UNICODE a 2 byte. Per ogni carattere usiamo 2 byte in memoria. Per operare su file di testo dobbiamo utilizzare stream di caratteri, così da assicurarci di fare la codifica giusta. Per invece file binari usiamo stream di byte (es. il file doc, se lo apro con un editor di testo esso non contiene il testo di word, ma un sacco di informazini su paragrafo, capitoli ecc., tutte le info sono scritte in binario).

37.7.1 Il problema dell'Endianness (Ordine dei Byte):

L'ordine dei byte nella memorizzazione nella trasmissione ha un certo impatto. Se la trasmissione/memorizzazione inizia dal byte più significativo, per finire con quello meno significativo, si parla di big-endian; se invece la trasmissione/memorizzazione inizia dal byte meno significativo, per finire con quello più significativo, si parla di little-endian. Questi termini derivano da "I Viaggi di Gulliver", noto romanzo di Jonathan Swift. Nel romanzo, due popolazioni, abitanti delle isole di Lilliput e Blefescu, entrano in rivalità per il modo in cui bisognerebbe aprire le uova: rompendo l'estremità più grande (come propugnato dalla popolazione dei Big-Endian), o quella più piccola (come vogliono i Little-Endian). Quindi, se io ho, per esempio, un intero a 4 byte, il primo byte che trasporto è quello più significativo o quello meno significativo? Questo ha un impatto perché il modo in cui il ricevente riceve i byte decide se quest'ultimo sia costretto a riordinarli o meno. La maggior parte dei casi la trasmissione di rete è fatta con little-endian, così che utilizzando delle code in ingresso il ricevente si trova l'ordine corretto in ingresso.

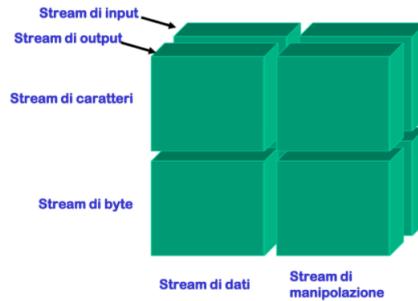
37.8 Stream di dati e di manipolazione:

Oltre agli stream di dati che collegano un programma con una sorgente o una destinazione abbiamo gli stream che elaborano dati in ingresso o in uscita. In Java possiamo definire sopra uno stream un altro stream. Questo stream sovrastante è detto di elaborazione. Lo stream di elaborazione lavora sullo stream di dati sottostante (es. la println non è un'operazione di base per uno stream di comunicazione, poiché uno stream di comunicazione di base scrive un byte. println invece scrive una riga. Per scrivere una riga abbiamo bisogno di uno stream aggiuntivo che lavori su quello di base). Il compito degli stream di manipolazione è quindi quello di manipolare i dati per poterli

poi scrivere correttamente sullo stream di dati. Anche gli stream di elaborazione si dividono in stream di input o di output e in stream di byte o di caratteri. Quindi in tutto abbiamo tre tipi di categorizzazione:

- Per direzione: input o output
- Per tipo di dati: byte o caratteri
- Per scopo: dati o manipolazione

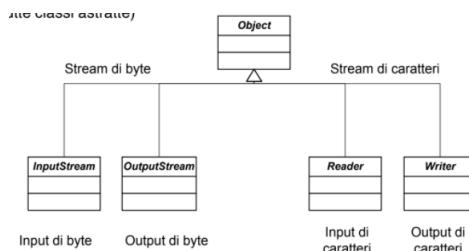
Queste tre classificazioni sono indipendenti, cioè combinabili in qualsiasi maniera. Abbiamo quindi in tutto 8 categorie, ecco una loro rappresentazione grafica tridimensionale: è da notare che le



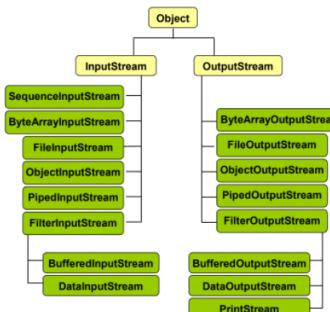
classi stream sono state realizzate per potersi incastrare l'una con l'altra: es. uno stream di manipolazione prende in input un altro stream. Inoltre il meccanismo degli stream è estendibile: possiamo creare i nostri stream di manipolazione (per motivi ad esempio di una particolare codifica di caratteri, oppure vogliamo un canale cifrato), quindi creiamo la nostra personale classe per lo streaming.

37.9 La Gerarchia delle Classi Stream:

Partiamo come al solito da Object. La gerarchia è contenuta nel package java.io e ha 4 classi derivate: Vediamo che non si palesa la differenza tra stream di dati e di manipolazione, perché



è una suddivisione concettuale, non interessa più a questo livello. Nella maggior parte dei casi in questo corso usiamo Reader o Writer. Ok, adesso però diamo un'occhiata alla gerarchia degli stream di Byte, quindi alle due classi InputStream e OutputStream. Vediamo delle classi per



esempio:

- FileInputStream e FileOutputStream per scrivere e leggere byte da file

- ObjectInputStream e ObjectOutputStream per memorizzare e prendere un'istanza di un oggetto in memoria. Procedura utile per scambiare oggetti in memoria tra diverse applicazioni.
- BufferedOutputStream, derivata da FilterOutputStream; e BufferedInputStream, derivata da FilterInputStream. Molto spesso su stream di comunicazione, specialmente su stream di rete, non si scrive carattere per carattere: si accumula un blocco di dati e si manda tutto insieme, in modo che si risparmiano dati. Questo è possibile grazie a queste due classi.

37.10 InputStream:

InputStream è una classe astratta che definisce pochi metodi:

- read(), ha il compito di leggere un carattere; quindi è lo stream di base di cui parlavamo prima. Astratto.
- available(), ha il compito di dire quanti caratteri sono disponibili in uno stream³⁷. Già implementato: restituisce sempre 0.
- close(), ha il compito di chiudere il canale. Già implementato: non fa niente.

Nota che tutti e tre i metodi lanciano IOException, cioè hanno `throws IOException` dopo la parentesi tonda, quindi se io uso questi metodi devo - siccome IOException sono di tipo catched - gestire le eccezioni in qualche modo. Le classi specifiche, che abbiamo visto sopra nella gerarchia, specializzano questi metodi, facendogli fare quello che devono fare (potremmo anche farle noi delle classi che specializzano questi metodi, ma in questo corso non lo faremo).

37.11 OutputStream:

OutputStream è una classe astratta che definisce pochi metodi:

- write(), equivalente di read(), scrive byte per byte (nota che, non avendo il tipo di dato byte, java rappresenta i byte come interi che vanno da 0 a 255). Astratto.
- flush(). Gli stream di dati non sono istantanei nella maggior parte dei casi; quindi per far scrivere sullo stream dobbiamo usare un flush, cioè scaricare³⁸. Già implementato: non fa niente.
- close(), ha il compito di chiudere il canale. Già implementato: non fa niente.

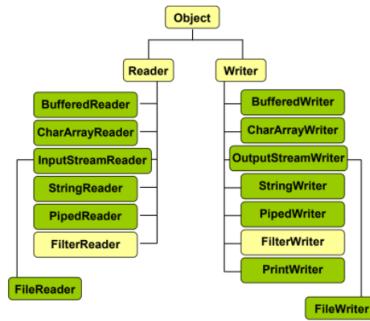
Nota che tutti e tre i metodi lanciano IOException, cioè hanno `throws IOException` dopo la parentesi tonda, quindi se io uso questi metodi devo - siccome IOException sono di tipo catched - gestire le eccezioni in qualche modo. Le classi specifiche, che abbiamo visto sopra nella gerarchia, specializzano questi metodi, facendogli fare quello che devono fare (potremmo anche farle noi delle classi che specializzano questi metodi, ma in questo corso non lo faremo).

38 Stream di Caratteri:

Si occupano automaticamente della internazionalizzazione, gestione lettere accentate e codifica. Ecco la gerarchia: BufferedReader permette di leggere una riga intera, sennò altrimenti il reader legge un carattere soltanto. PrintWriter contiene la println, quindi tutte le cose che contengono println sono generalmente estensione di PrintWriter. Poi abbiamo FileReader e FileWriter per leggere e scrivere da file di testo

³⁷ Lo stream resta aperto finché non lo chiudi esplicitamente, però potrebbero non esserci dati. Esempio: sto leggendo da tastiera, console aperta ma utente non sta scrivendo nulla, quindi i caratteri available() sono 0, appena inizia a scrivere l'utente sono maggiori di 0. Quindi ci sarà un ciclo while su available() e si andrà a leggere carattere per carattere non appena arrivano i caratteri.

³⁸ Questo risponde in generale alla domanda, perché dobbiamo fare l'espulsione quando stacco la pennetta USB? Perché in questo modo impone un flush() a tutti gli stream aperti sulla pennetta. Questo perché è possibile che alcune scritture che abbiamo fatto non siano ancora state caricate sulla periferica, perché per performance si preferisce farli piano piano. L'espulsione forza tutti gli stream a fare flush(), cioè a scaricare tutti i dati dall'area temporanea del Sistema Operativo in cui sono all'area dove devono essere scritti.



38.1 Reader:

Reader è una classe astratta che definisce pochi metodi:

- `read()`, legge carattere per carattere, restituisce un intero quindi bisogna fare cast esplicito (l'int sarebbe il codice della codifica corrispondente al carattere). Astratto.
- `ready()`, equivalente a `available()`: restituisce un booleano che dice se c'è qualcosa da leggere o meno. Già implementato: restituisce sempre 0.
- `close()`, ha il compito di chiudere il canale. Già implementato: non fa niente.

Nota che tutti e tre i metodi lanciano `IOException`, cioè hanno `throws IOException` dopo la parentesi tonda, quindi se io uso questi metodi devo - siccome `IOException` sono di tipo catched - gestire le eccezioni in qualche modo. Le classi specifiche, che abbiamo visto sopra nella gerarchia, specializzano questi metodi, facendogli fare quello che devono fare (potremmo anche farle noi delle classi che specializzano questi metodi, ma in questo corso non lo faremo).

38.2 Writer:

Writer è una classe astratta che definisce pochi metodi:

- `write(int c)`, scrive carattere per carattere, prende un intero quindi bisogna fare cast esplicito (l'int sarebbe il codice della codifica corrispondente al carattere). Astratto.
- `write(String str)`, overload() del `write` precedente, scrive una stringa (in realtà prende l'intero codice della codifica dell'intera stringa e chiama `write(int c)` mettendo come parametro questo intero che ha calcolato). Astratto.
- `flush()`. Già implementato: non fa niente.
- `close()`. Già implementato: non fa niente.

Nota che tutti e tre i metodi lanciano `IOException`, cioè hanno `throws IOException` dopo la parentesi tonda, quindi se io uso questi metodi devo - siccome `IOException` sono di tipo catched - gestire le eccezioni in qualche modo. Le classi specifiche, che abbiamo visto sopra nella gerarchia, specializzano questi metodi, facendogli fare quello che devono fare (potremmo anche farle noi delle classi che specializzano questi metodi, ma in questo corso non lo faremo).

39 I/O Standard:

Per ora abbiamo usato `System.in` e `System.out` come scatole nere, ora cerchiamo di capirle bene. Sappiamo che queste due sono variabili statiche della classe `System`. Servono per gestire input da tastiera e output su video solo da console (per le interfacce grafiche infatti non vengono usate queste due variabili). Purtroppo per ragioni storiche in Java 1.0³⁹ non c'erano stream di caratteri, e per ragioni di retrocompatibilità `System.in` e `System.out`, che esistevano già, non sono state modificate (sennò si rompe tutto). Perciò `System.in` e `System.out` sono stream di byte, in particolare:

³⁹La prima versione di Java utilizzabile in ambito professionale senza bestemmiare è la Java 1.4, o, utilizzando la notazione corrente, Java 4.0.

- System.in è di tipo InputStream, e non mette a disposizione metodi per scrivere qualunque tipo di dato ma fornisce solo servizi di base.
- System.out è di tipo PrintStream, e mette a disposizione i metodi print() e println() che consentono di scrivere a video qualunque tipo di dato.

Quindi System.out lo usi direttamente mentre System.in mai direttamente.

Gestione della Tastiera: come gestiamo allora System.in, dato che, essendo come detto uno stream di byte e non possedendo metodi per leggere comodamente un'intera stringa, non consente di suo di trattare in modo corretto l'input? Usiamo il meccanismo degli incastri di Java: usiamo due classi che discendono da Reader, InputStreamReader e BufferedReader. Vediamo quindi che si tratta di due esempi di stream di manipolazione, che manipolano altri stream, e non vengono usati direttamente sulla periferica:

- InputStreamReader è una classe di Reader che permette di prendere in input un InputStream e farlo diventare un Reader di caratteri, che legge carattere per carattere.
- BufferedReader è una classe sempre di Reader il cui costruttore prende in input un'istanza di InputStreamReader (che come appena detto fa lettura carattere per carattere).

Quindi, in Java scriveremmo:

```
public MainApp {
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); //InputStream
        StreamReader prende uno stream di byte e lo trasforma in uno stream di
        caratteri, poi diamo questo InputStreamReader come input a BufferedReader,
        come già detto

        try {
            br.readLine(); //BufferedReader un reader bufferizzato quindi permette di
            leggere una riga intera insieme e non solo carattere per carattere, mentre
            InputStreamReader, seppur stream di Caratteri e non di byte comunque
            questi caratteri li legge uno alla volta
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Questa è la soluzione a basso livello, più efficiente. Noi però utilizziamo la soluzione che abbiamo già utilizzato, decisamente meno efficiente però: usiamo la classe Scanner, offerta dalla libreria standard di Java nel package java.util. Scanner converte lo stream di byte in caratteri, prendendo in input un InputStream (lo può prendere da un File, o derivato da una Socket ...). I metodi di Scanner che permettono questo sono:

- nextLine(): restituisce in forma di oggetto String la successiva riga d'ingresso data in input, cioè tutti i caratteri fino al tasto Enter ("\n");
- next(): restituisce in forma di oggetto String la successiva parola fornita in ingresso, cioè tutti i caratteri fino al carattere di spaziatura.
- nextDouble(): restituisce in forma di dato double il numero reale fornito in ingresso.
- nextInt(): restituisce in forma di dato int il numero intero fornito in ingresso.

Come già detto, in realtà lo Scanner è meno efficiente della soluzione che mette in pipeline stream di input e stream di manipolazione bufferizzati⁴⁰, ma a noi ci va più che bene.

⁴⁰Se vuoi vedere perché è meno efficiente: <http://www.davismol.net/2015/04/27/java-io-bufferedReader-e-fileInputStream-vs-scanner-confronto-su-lettura-e-parsing-di-un-file-da-200k-linee/>

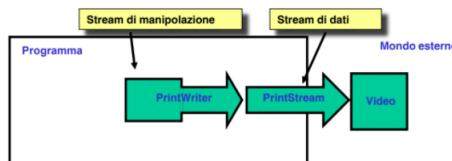
Gestione del Video: come già detto, a differenza di System.in, System.out è già sufficiente per gestire l'output e non abbiamo bisogno di soluzioni come prima. Quindi lo usiamo direttamente. Tuttavia volendo possiamo usare una tecnica simile a quanto abbiamo fatto sopra per System.in: ci basta usare un solo stream di manipolazione stavolta, PrintWriter (sottoclasse di Writer), che mette a disposizione anche lui i metodi print() e println() e definisce questo costruttore:

```
public PrintWriter(OutputStream out)
```

Potremmo poi scrivere:

```
PrintWriter video = new PrintWriter(System.out),
video.println(12);
video.println( Ciao );
video.println(13,56);
```

Quindi prima creando un'istanza di PrintWriter e poi usando println sul PrintWriter nello stesso modo con cui useremmo System.out direttamente. Ma se andava been System.out direttamente



perché applicargli questso stream di manipolazione? Perché System.out gestisce male i caratteri orientali, quelli cirillici ecc. Ma, in realtà, come è ovvio, per gli usi che ne facciamo noi va più che bene usare direttamente System.out, come abbiamo fatto finora d'altronde.

40 File:

La classe File in Java rappresenta i file. Questa classe è una rappresentazione astratta di file e directory. Il concetto di file è un concetto gerarchico (pensa all'albero delle directory). Anche le directory sono dei File. In particolare i File file (cioè quelli che noi chiamiamo file normalmente) sono le foglie della gerarchia, mentre i file directory sono dei nodi. Sia per rappresentare file che directory si usa la classe File. Questa classe contiene metodi statici per gestire la struttura dal file system: quindi eliminare, copiare ecc. directory e file. Non contiene però metodi per leggere o scrivere su file. Per fare queste due ultime azioni dobbiamo associare al file uno stream. Gli stream che usiamo noi sono:

- FileInputStream/FileOutputStream, che sono sottoclassi di InputStream e OutputStream (quindi sono stream di byte). I costruttori sono (prendiamo per esempio FileInputStream):

```
public FileInputStream(File file) throws FileNotFoundException
public FileInputStream(String name) throws FileNotFoundException
```

Vediamo che abbiamo due costruttori: uno prende in input un file, che si crea usando il costruttore File() e dando in input a quest'ultimo il percorso del file; l'altro costruttore prende in input direttamente il percorso del file, in forma di stringa. Entrambi i costruttori lanciano l'eccezione FileNotFoundException. Notare che si possono applicare dei filtri a FileInputStream e a FileOutputStream, ad esempio DataInputStream e DataOutputStream.

- FileReader/FileWriter, che sono sottoclassi di Reader e Writer (quindi sono stream di caratteri). I costruttori sono:

```
public FileReader(File file) throws FileNotFoundException
public FileReader(String name) throws FileNotFoundException
public FileWriter(File file) throws FileNotFoundException
public FileWriter(String name) throws FileNotFoundException
public FileWriter(String name, boolean append) throws FileNotFoundException
```

Vediamo che l'ultimo costruttore di FileWriter ha un booleano che specifica se la scrittura sul file appende al suddetto o lo sovrascrive. Cioè, se mettiamo append uguale a true allora scrivendo su un file le cose nuove verranno aggiunte alle fine, mettendo uguale a false le cose nuove sovrascriveranno l'intero contenuto del file. Il professore non si ricorda se, nel caso degli altri costruttori, in cui non abbiamo la variabile append, abbiamo di default append a true o a false. Notare che si possono applicare BufferedReader e BufferedWriter a FileReader e a FileWriter.

40.1 Lettura di un File di Testo, Esempio:

Supponiamo di voler leggere il seguente file di testo, chiamato inventory.dat:

```
Quaderno 14 1.35
Matita 132 0.32
Penna 58 0.92
Gomma 28 1.17
Temperino 25 1.75
Colla 409 3.12
Astuccio 142 5.08
```

Quindi ogni riga del file è un prodotto, con accanto nome, quantità e prezzo unitario, separati da spazi. Il metodo classico di lavorare sui file in Java prevede che si crei una classe che descrive ogni record nel file, e infatti quello che facciamo in questo caso è che i dati letti dal file sovrastante vengono messi in un array di oggetti di classe InventoryItem. La classe InventoryItem è definita così:

```
public class InventoryItem
{
    private String name;
    private int units;
    private float price;

    public InventoryItem(String nm, int num, float pr)
    {
        name = nm; units = num; price = pr;
    }

    public String toString()
    {
        return name + ": " + units + " a euro " + price;
    }
}
```

Quindi un'istanza di Inventory Item ha un nome, un'unità e un prezzo unitario, e possiamo stampare queste sue informazioni attraverso l'override del `toString()`. Ok, ma come li leggiamo sti dati? Innanzitutto notiamo che parliamo di lettura di file di testo, quindi usiamo stream di caratteri. La metodologia è questa: leggiamo riga per riga e applichiamo un tokenizer per prendere i singoli elementi. Poi questi singoli elementi vengono trasformati per entrare in un'istanza di InventoryItem.

40.2 StringTokenizer:

Riprendiamo il nostro file:

```
Quaderno 14 1.35
Matita 132 0.32
Penna 58 0.92
Gomma 28 1.17
Temperino 25 1.75
Colla 409 3.12
Astuccio 142 5.08
```

All'interno di ogni riga del file che dobbiamo leggere abbiamo diverse informazioni, separate da spazi. Dobbiamo quindi scomporre ogni riga per ottenere i singoli elementi in essa contenuta. Per fare ciò usiamo la classe StringTokenizer, inclusa nel package java.util. Il costruttore del Tokenizer prende in input la stringa da scomporre (ed eventualmente il carattere di tokenizzazione, ma nel nostro caso è lo spazio, e questo viene usato come carattere di tokenizzazione di default, quindi non inseriamo questo secondo parametro) e il metodo nextToken() ci permette di andare avanti nella riga estraendo le singole sottostringhe, per poi farci quello che vogliamo, nel nostro caso farle diventare i campi di un'istanza di InventoryItem. Sapendo questo ora, possiamo pensare di aggiornare la nostra classe InventoryItem in questa maniera:

```

public class InventoryItem
{
    private String name;
    private int units;
    private float price;

    public InventoryItem(String nm, int num, float pr)
    {
        name = nm; units = num; price = pr;
    }

    public InventoryItem(String linea) {

        StringTokenizer st = new StringTokenizer(linea) //non mettiamo il carattere di
                                                       //tokenizzazione cos che di default esso sia " "

        this.name = st.nextToken();
        this.units = Integer.parseInt(st.nextToken());
        this.price = Float.parseFloat(st.nextToken());

    }

    public String toString()
    {
        return name + ": " + units + " a euro " + price;
    }
}

```

Ok, fatto StringTokenizer andiamo a fare la vera e propria lettura da file. Per farlo usiamo Scanner:

```

public Main {

    public Logger logger = Logger.getLogger("loggerNumero1");

    public static void main(String[] args) {

        try {

            Scanner sc = new Scanner(new File("src/inventory.dat")) //il file che gli
                                                               //passiamo deve esistere veramente! Se il file non esiste viene generata
                                                               //l'eccezione FileNotFoundException. Questa un'eccezione di tipo
                                                               //IOException e quindi catched: va gestita.

            while (sc.hasNext()) {

                String linea = sc.nextLine();
                InventoryItem ii = new InventoryItem(linea);

                logger.log(level.info, ii.toString()); //passiamo come messaggio da
                                                       //stampare nel log il toString, e non direttamente ii, perch? Perch log,
                                                       //se gli passi solo ii come secondo parametro, non chiama direttamente
                                                       //il toString per stampare ii (come invece println(ii) avrebbe fatto), e
                                                       //quindi bisogna esplicitamente passargli il toString() di ii
            }
        }
    }
}

```

```

        }

        sc.close();

    } catch (FileNotFoundException e) {
        logger.log(Level.SEVERE, "File non trovato", e.getMessage()); //gli passiamo
            come parametro e.getMessage() che sulla classe exception permette di
            ottenere il messaggio dell'eccezione.
    }

}

```

E questo è come leggere i file con il semplice Scanner. Se, senza aver creato effettivamente il nostro file inventory.dat, proviamo ora a runnare il programma, vedremo in console il messaggio di log "File non trovato". Quindi dobbiamo mettere il file nella giusta directory al fine di farla leggere correttamente da NetBeans. Come nel caso del logging.properties, o mettiamo il file nella directory principale dell'applicazione (cioè NetBeansProject -> NomeDelProgetto) oppure, come abbiamo fatto noi nel caso del logging.properties e come faremo anche qua, mettiamo nel costruttore di scanner, precisamente nel costruttore del file il percorso relativo, a partire da una certa directory, e nel nostro caso mettiamo src/, così da poter inserire facilmente il nostro file da IDE semplicemente creando il file premendo il tasto destro su Source Packages. Ora eseguendo il programma verrà stampato su console l'intero file ma diviso in oggetti dal tokenizer di InventoryItem. Nota che se vogliamo possiamo aggiungere una struttura dati che contenga ogni InventoryItem facilmente.

Ok, e questo era con Scanner. Ma noi sappiamo che c'è un metodo meno semplice per fare lettura da file, utilizzando FileReader e BufferedReader.

```

public Main {

    public Logger logger = Logger.getLogger("loggerNumero1");

    public static void main(String[] args) {

        try {
            BufferedReader br = new BufferedReader(new FileReader(new File("inventory.dat")));
            String nuovaRiga = br.readLine();

            while (nuovaRiga != null) { //non abbiamo hasNext() per buffered reader, ma
                sappiamo che restituisce null quando non ci sono pi righe da leggere, quindi
                impostiamo cos il ciclo

            InventoryItem ii = new InventoryItem(nuovaRiga);

            logger.log(Level.INFO, ii.toString()); //anche qui, passiamo ii.toString() e
                non direttamente ii perch log, a differenza di println, non chiama
                direttamente il toString se passiamo l'oggetto ii

            nuovaRiga = br.readLine();
        }

    } catch (FileNotFoundException e) {
        logger.log(Level.SEVERE, "File non trovato", e.getMessage());
    } catch (IOException e) { //perch readLine() throws IOException
        logger.log(Level.SEVERE, "Errore di lettura", e.getMessage());
    }
}

```

}

40.3 Scrrittura di un file di Testo, esempio:

Qui non abbiamo utility come Scanner, ma usiamo direttamente gli stream. Usiamo FileWriter (ma questo ci fa scrivere carattere per carattere), e gli agganciamo PrintWriter (ci fa scrivere per riga). Facciamo un programma che scrive su un file la tavola pitagorica.

```
public Main {  
  
    PrintWriter pw = new PrintWriter(new FileWriter(New File("src/pythaTable.txt")));  
  
    public static void main(String[] args) {  
        for (int i = 0; i <= 10; i++) { //Pythagorean table  
            for (int j = 0; j <= 10; j++) {  
                pw.print(i*j + "\t");  
            }  
            pw.println("");  
        }  
        pw.close();  
    } catch (IOException e) {  
        Logger.getLogger(Main.class.getName()).log(Level.SEVERE, "Errore di lettura",  
            e); //questo il catch creato direttamente dall'IDE tramite l'opzione  
            della lampadina, vediamo che crea un nuovo logger identificato col nome  
            della classe  
    }  
}
```

Part X

Sockets e rudimenti di programmazione di rete:

All'esame noi creiamo un client che si collega a un computer, in cui i prof faranno girare un programma.

41 Rete di Calcolatori:

Una rete di calcolatori è un sistema che permette la condivisione di dati e risorse tra diversi calcolatori. Può essere privata o pubblica⁴¹. All'interno di una rete di calcolatori la comunicazione avviene tramite messaggi, che scambiano informazioni. La maggior parte delle reti che usiamo oggi sono a instradamento di pacchetto, cioè qualunque messaggio viene diviso in piccoli pacchetti, che seguono delle strade indipendenti l'uno dall'altro all'interno della rete per arrivare all'altro calcolatore. Fanno strade indipendenti perché ci sono Router intermedi che prendono questi pacchetti e li indirizzano. Per inviare un messaggio a un altro calcolatore è necessario un canale fisico (cavo telefonico, fibra ottica, onde radio ...) ⁴²) e un protocollo di comunicazione (cioè un linguaggio comune): si tratta di un insieme di regole formali, noi utilizziamo TCP/IP⁴³. Dopo tutto sto discorso sulle reti di calcolatori, quello che ci interessa per gli obiettivi di questo corso è semplicemente **mettere in comunicazione due computer con una connessione affidabile, per scambiarsi dei dati.**

41.1 Paradigmi di Comunicazione:

- Client-Server: le applicazioni di rete hanno due programmi distinti, il client e il server. Il client è in esecuzione su un elaboratore (host), il server sull'altro (l'altro host). Il server si mette in attesa di una richiesta da servire, mentre il client fa la richiesta. Tipicamente un client comunica con un solo server, mentre un server comunica con più client contemporaneamente. Esempio: se cerco una cosa su google il server è il sistema di Google (i suoi calcolatori da qualche parte nel mondo), mentre il client è browser che usiamo.
- Peer-to-peer: più host comunicano tra loro comportandosi sia da client che da server.

Noi ci concentriamo sul paradigma client-server, in particolare all'esame facciamo un applicazione client e il server ci viene fornito, come già detto.

41.2 Identificazione dei nodi:

In una rete basata su IP ogni computer è identificato da un indirizzo IP a 32 bit⁴⁴. Questi indirizzi sono difficili da ricordare, quindi si usano dei nomi, suddivisi in livelli (es. www.google.it). I DNS

⁴¹Internet è la più grande rete pubblica del mondo, ed è una rete globale. La rete di calcolatori del laboratorio Antonio Ruberti a via Tiburtina è una rete locale. Possiamo dividere le reti di calcolatori in: PAN (personal area network, quindi il tuo smartphone e le sue periferiche, come le cuffiette), LAN (local area network, quindi una rete a livello di edificio), WAN (wide area network, le reti a livello cittadino), e infine Internet, che si presenta quindi come una rete di reti.

⁴²La maggior parte del traffico dati dall'Europa all'America avviene tramite dei cavi cablati in fibra ottica che attraversano l'Atlantico: non usiamo satelliti perché sono lenti, e quindi c'è un forte ritardo di comunicazione (chiamato oggi ping).

⁴³IP è il protocollo che permette la comunicazione da un computer a un altro computer. TCP lavora sui servizi: dopo aver messo in comunicazione i due computer, devo dire quale servizio del computer a cui mi sono collegato mi serve, per fare questo devo instaurare una connessione, precisamente instauro una connessione affidabile. Con TCP si fanno connessioni affidabili, con UDP si fanno connessioni non affidabili, ma non vediamo UDP in questo corso. Le connessioni affidabili sono quelle in cui non si perdono pacchetti, in quelle non affidabili (quelle che usiamo tutti i giorni, quando facciamo una chiamata whatsapp o entriamo in una stanza zoom) invece c'è il rischio di perdere pacchetti.

⁴⁴In realtà l'indirizzo IP identifica una scheda di rete. Se un computer ha più schede di rete allora è identificato da più IP. Un indirizzo importante è ad esempio l'indirizzo 127.0.0.1, che è il localhost, cioè è l'indirizzo IP dello stesso computer su cui stiamo operando. Questo indirizzo non è riservato, quindi non si può raggiungere da altre macchine. Un altro indirizzo importante è 0.0.0.0, che indica tutte le schede (o interfacce) di rete del calcolatore, perché ricordiamo che ogni computer non ha un solo indirizzo IP ma ha un indirizzo IP per ogni sua scheda di rete.

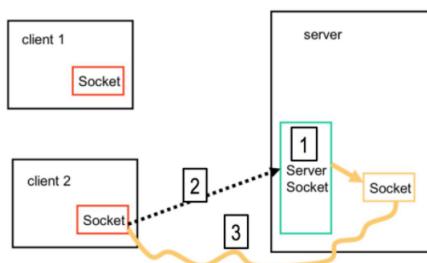
(Domain Name Server) si occupano di convertire i nomi in indirizzi⁴⁵.

41.3 Servizi e Sever:

Ogni server è collegato ad una rete e ha quindi un indirizzo. Ad ogni servizio del server è associato un numero di porta, così che questo sia identificato dall'indirizzo del server e dal numero della porta. Ad un servizio erogato corrispondono dei software in esecuzione (processi) che sono in "ascolto" sulla porta corrispondente, aspettando un client che richieda un servizio.

42 Socket:

Il socket permette di stabilire un canale di comunicazione con un altro processo, che può essere sulla stessa macchina oppure un'altra. Quello che viene trasmesso sono pacchetti TCP. In Java usiamo il package java.net, che ha le classi Socket (identifica canale di comunicazione tra due macchine, su un socket possiamo richiedere un inputstream e un outputstream. Cioè i socket sono bidirezionali (ma poiché gli stream sono monodirezionali abbiamo bisogno di due stream). Stiamo parlando di stream di byte.) e ServerSocket (permette di creare il server, perché permette di mettere il programma in attesa di connessioni con client. Una volta che il client si è connesso viene instaurato il canale di comunicazione. Quindi in pratica il ServerSocket ci permette di dire al mondo "sto esponendo questo indirizzo su indirizzoIP:Porta, e sono aperto a connessioni"). Vediamo un esempio:



1. il ServerSocket del server si mette in attesa di una connessione
2. il Socket del client si connette al ServerSocket
3. Viene creato un Socket nel server e quindi stabilito un Canale di Comunicazione tra server e client.

Come lo facciamo in Java?

- Come mettersi in attesa (lato server):

- creo un'istanza di ServerSocket passando porta e opzionalmente indirizzo ip (se non passo l'indirizzo ip si suppone 0.0.0.0, quindi si suppone che la porta sia aperta su tutte le mie interfacce di rete).

```
ServerSocket serverSocket = new ServerSocket(4567);
```

- chiamo il metodo accept(), che è bloccante: blocca il programma fin quando un client non si connette.

```
Socket socket = serverSocket.accept();
```

- Una volta che qualcuno si connette lato server accept() termina la sua esecuzione e viene restituita al server un'istanza di Socket. Quindi quando viene instaurata una connessione client-server sia il client che il server hanno una propria istanza di Socket, ed entrambe fanno riferimento allo stesso canale di comunicazione.

⁴⁵Quando nel 2021 tutti i servizi social dell'azienda Facebook (ora Meta) andarono in down, il software di correzione automatica del codice aveva sballato tutte le voci DNS, quindi ogni utente, provando a inserire il nome (es. www.facebook.it) non riusciva a connettersi al server perché i DNS non riuscivano a tradurre questo nome nell'indirizzo IP corrispondente.

- Come aprire una connessione (lato client):
 - creo un’istanza di Socket, specificando IP e porta del server. Nel nostro caso avevamo messo come numero di porta del server 4567, e non avendo specificato un IP vanno bene tutti quelli corrispondenti alle interfacce di rete del server. Scegliamo come IP 127.0.0.1, a cui il client può accedere poiché supponiamo che sia il programma client che il programma server siano sullo stesso calcolatore (ricordiamo che 127.0.0.1 corrisponde al local host, cioè è l’IP della propria macchina, quella su cui si sta operando).

```
Socket socket = new Socket("127.0.0.1", 4567)
```

- all’indirizzo e porta specificati ci deve essere già un server in ascolto.

Se la connessione ha successo sia il client che il server possono usare gli stream associati ai due sockets per leggere e scrivere. Precisamente:

- per leggere usiamo getInputStream(). Ma getInputStream() dà uno stream di Byte, quindi ci agganciamo Scanner che si occupa della conversione in stream di caratteri.
- per scrivere usiamo getOutputStream(), ma dato che si tratta di uno stream di byte ci applichiamo PrintWriter.

42.1 Esercizio, Echo Server:

Si crei un server che accetta connessioni TCP sulla porta 1337. Una volta accettata la connessione il server leggerà ciò che viene scritto una riga alla volta e ripeterà nella stessa connessione ciò che è stato scritto. Se il server riceve una riga “quit” chiuderà la connessione e terminerà la sua esecuzione.

```
package echoserver;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Client {
    public static void main(String[] args) {
        try {
            Socket socket = new Socket("127.0.0.1", 1337);
            PrintWriter pw = new PrintWriter(socket.getOutputStream());
            Scanner scannerDaConsole = new Scanner(System.in); //ci serve per memorizzare
            in nuovaLinea la stringa che poi spediamo al server

            while(true) { //al posto di true andava bene anche scannerDaConsole.hasNext()
                che d true finche non si fa scanner.close()

                String nuovaLinea = scannerDaConsole.nextLine();
                pw.println(nuovaLinea); //scrive la linea nel server
                pw.flush();
            }
        } catch (IOException ex) {
            Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

```

        if (nuovaLinea.equals("quit")) {
            break;
        }
    }

} catch (IOException ex) {
    Logger.getLogger(Client.class.getName()).log(Level.SEVERE, null, ex);
}

}

-----
package echoserver;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {

    public static void main(String[] args) {

        try { //ServerSocket's constructor throws IOException, a catched exception

            ServerSocket ss = new ServerSocket(1337);

            System.out.print("In attesa di connessione... ");
            Socket listeningSocket = ss.accept();
            System.out.println("connesso!");

            Scanner scanner = new Scanner(listeningSocket.getInputStream());

            while(true) { //al posto di true andava bene anche scanner.hasNext() che d
                true finche non si fa scanner.close()

                String nuovaLinea = scanner.nextLine();

                if (nuovaLinea.equals("quit")) {
                    break;
                } else {
                    System.out.println(nuovaLinea);
                }
            }

        } catch (IOException ex) {
            Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
        }

    }
}

```

Quindi abbiamo creato due classi, in due file, ognuno con un main. Nota che in generale client e server sono due progetti differenti, quindi non nello stesso progetto. Vediamo che in questo semplice esempio il server può avere al massimo un client connesso, quando vedremo i thread scopriremo come connetterne più di uno. Inoltre vediamo che in questo caso il client scrive soltanto nel server e il server legge soltanto dal client, quindi non c'è palleggio di informazioni, anche se potremmo dato

che i socket sono bidirezionali. Ultima osservazione: `pw.flush()` se non lo facciamo manualmente dobbiamo aspettare che lo faccia il PrintWriter autonomamente, e ci può mettere un po' di tempo.

42.2 Esercizio Lotto:

Si vuole realizzare un'applicazione client/server che permette di giocare ad una variante del "Gioco del lotto". Lato client, l'utente inserisce 5 numeri (da 0 a 9) ed avvia l'estrazione da parte del server. Il server genera 5 numeri casuali da 0 a 9, uno per ogni casella. Due caselle differenti possono contenere due numeri uguali. Il server è multithreading ed accetta connessioni da più client. I client, una volta connessi, possono avviare una sessione di gioco, come di seguito specificato. Una volta avviata la sessione, il server invierà stringhe indicanti la posizione della casella e il numero estratto. Le stringhe inviate dal server sono elaborate dal client secondo una logica descritta in seguito. Due client connessi allo stesso server gestiscono partite differenti. Durante la trasmissione delle stringhe, il client può decidere di interrompere la ricezione in qualsiasi momento. La comunicazione è basata unicamente su scambio di stringhe. Tutte le stringhe sono inviate da client a server e viceversa utilizzando il carattere di fine linea come separatore.



Si richiede la realizzazione del client, con interfaccia grafica e networking, in grado di comunicare con un server multithreading (fornito).

L'interfaccia grafica del client (si consiglia 700x200 come risoluzione) dovrà essere composta da un frame che abbia per titolo *nome cognome e matricola* dello studente, da due campi testuali (l'indirizzo IP e la porta a cui connettersi che verranno comunicati in aula), da cinque pulsanti (*Connect*, *Disconnect*, *Start*, *Interrompi* e *Clear*) che permettono rispettivamente di gestire la connessione col server, di disconnettersi dal server, di avviare/interrompere una sessione di gioco e di ripristinare il contenuto ed il colore delle 5 caselle. Inoltre, l'interfaccia grafica dovrà essere composta da 1 pannello che a sua volta contiene 5 istanze di *ColoredButton* (caselle) disposte in una griglia 1*5 come in figura.

All'avvio le caselle devono essere tutte del colore *LIGHT_GRAY*. L'utente deve poter cliccare le caselle per inserire dei numeri da estrarre da 0 a 9. Il click su una casella è già gestito nella classe *ColoredButton*, fornita funzionante. Lo studente, tuttavia, dovrà implementare il metodo *read()* richiamato all'interno della classe *ColoredButton*, per leggere da input una cifra compresa tra 0 e 9. (Suggerimento: si veda la classe *NumberInput* e la relativa interfaccia *Reader*). Se la cifra non è compresa tra 0 e 9, la lettura dovrà essere ripetuta affinché tale condizione sia rispettata.

Si richiede di gestire correttamente la possibilità di premere i pulsanti, in particolare:

- All'avvio solamente i pulsanti *Connect* e *Clear* sono abilitati. Non deve essere possibile avviare 2 connessioni dallo stesso client spingendo ripetutamente il bottone *Connect*.
- *Start* e *Disconnect* possono essere premuti solo se la connessione col server è già avviata.
- Dopo la pressione di *Start* l'unico pulsante che può essere premuto è *Interrompi*.
- Alla pressione di *Interrompi* dovranno tornare attivi solo i pulsanti *Start*, *Disconnect* e *Clear*, per effettuare un nuovo gioco/estrazione, disconnettersi dal server, e ripristinare le 5 istanze di *ColoredButton*.
- *Clear* può essere premuto solo se non si stanno ricevendo stringhe dal server.

Si implementi il seguente protocollo:

- Alla pressione del pulsante *Connect*, il client invierà una richiesta di connessione al server utilizzando Indirizzo IP e porta indicati negli appositi campi.
- Alla pressione del tasto *Start*, dopo la connessione, il client deve controllare che l'utente abbia inserito 5 numeri da estrarre. In caso positivo, deve mandare il comando "start" al server, abilitare inoltre il pulsante *Interrompi*, e disabilitare *Start*, *Disconnect* e *Clear*. Il server inizierà ad inviare ad intervalli regolari stringhe nel seguente formato: "posizione_griglia;numero_estratto". Esempio: 3:4. Il dominio del campo *posizione_griglia* è ('0', '1', '2', '3', '4'), mentre quello del campo *numero_estratto* è ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9').

Durante l'estrazione, il client alla ricezione della stringa dovrà cambiare colore alla casella nel pannello alla posizione indicata da *posizione_griglia*. Il colore diventerà Color.GREEN se la casella è vincente, Color.RED altrimenti. Una casella alla posizione *posizione_griglia* è vincente se *numero_estratto* per quella casella è uguale a quello che l'utente ha inserito nella fase precedente all'estrazione. Durante l'estrazione (quando si preme *Start*) il click sulle caselle dovrà essere disabilitato.

L'estrazione da parte del server termina quando quest'ultimo, dopo aver estratto il quinto numero invia una stringa che avrà *posizione_griglia* = "*" e *numero_estratto* = "*". Alla ricezione di questi due caratteri, il client dovrà smettere di ricevere stringhe, e comunicare, tramite un messaggio in popup il vincitore (Server o Utente). Il criterio di vittoria a fine partita, è il seguente:

```
if (n° delle caselle colorate Color.GREEN > 0) then User Wins;
else Server Wins;
```

Al termine dello scambio, le caselle dovranno rimanere colorate e il click su di esse dovrà essere nuovamente abilitato. Il client dovrà inoltre disabilitare il pulsante *Interrompi* e riabilitare *Start*, *Disconnect* e *Clear*.

Nel caso in cui si avvii una nuova comunicazione dopo l'interruzione (si prema di nuovo *Start*), preliminarmente il colore di ogni casella deve essere riportato a *LIGHT_GRAY* ed il contenuto delle caselle deve rimanere inalterato.

Come avviare il server? O si fa doppio click sul fil .jar (situato nella cartella drive del corso), e in alcuni casi funziona, altrimenti si apre un Prompt dei comandi e scriviamo:

```
java -jar nomedelServer.jar
```

Questo esegue il programma. Premendo start il server si mette in attesa di connessioni. Possiamo vedere cosa fa il server senza scrivere un client, precisamente usando il comando Telnet. Sempre da prompt dei comandi scriviamo:

(suggerimento: fare riferimento al metodo `changeColor(Color c)` della classe `ColoredButton` per i colori).
(suggerimento 2: fare riferimento al metodo `getDigit()` della classe `ColoredButton` per ottenere la cifra all'interno di una casella, inserita nella fase precedente all'estrazione)
(suggerimento 3: fare riferimento al metodo `isGreen()` della classe `ColoredButton` per verificare se la casella su cui viene richiamato tale metodo è di colore verde, così da determinare il vincitore della partita).

-
- Alla pressione del tasto `Interrompi`, il client deve inviare il comando “`interrompi`”. Il server risponderà inviando una stringa che avrà `posizione_griglia = '-1'` e `numero_estratto = '-1'`. Alla ricezione di questi due caratteri, il client dovrà smettere di ricevere stringhe, e comunicare, tramite un messaggio in popup, la sconfitta del client. L'interruzione dell'estrazione determina la sconfitta automatica del client. Al termine dello scambio, le caselle dovranno rimanere colorate e il click su di esse dovrà essere nuovamente abilitato. Il client dovrà inoltre disabilitare il pulsante `Interrompi` e riabilitare `Start`, `Disconnect`, e `Clear`.
 - Alla pressione del pulsante `Disconnect`, deve essere inviata la stringa “`disconnect`”, e devono essere chiusi i canali di comunicazione generati in fase di connessione. Deve inoltre essere abilitato nuovamente il pulsante `Connect` in quanto deve essere possibile instaurare una nuova connessione senza che sia necessario il riavvio del client.
 - Alla pressione del pulsante `Clear`, il colore di ogni casella deve essere riportato a `Color.LIGHT_GRAY` ed il contenuto ripristinato.
(suggerimento: fare riferimento al metodo `setTextDigit(String digit)` della classe `ColoredButton` per ripristinare il contenuto di una casella).

telnet 127.0.0.1 numerodellaporta

Dove numerodellaporta è il numero della porta messa nel server. Ora, sempre da questo prompt dei comandi, possiamo scrivere i comandi che avremmo scritto dal client e il server ci risponde sullo stesso terminale. In particolare se scriviamo start il server ci stamperà sul terminale i vari posizione:numeroestratto.

Bene, ora scriviamo il codice del nostro programma. Eccolo qua:

```
package client;

import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.Color;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class MyClientFrame extends JFrame {

    private static final String TITOLO = "Simone Palumbo 1938214";
    private static final int LARGHEZZA = 700;
    private static final int ALTEZZA = 200;

    final JPanel northPanel = new JPanel();
    final JButton startButton = new JButton("Start");
    final JLabel ipLabel = new JLabel("IP Address");
    final JTextField ipTextField = new JTextField("127.0.0.1");
    final JLabel portLabel = new JLabel("Port");
    final JTextField portTextField = new JTextField("4400");
    final JButton interrompiButton = new JButton("Interrompi");

    final JPanel centralPanel = new JPanel(new GridLayout(1,5));
    final ColoredButton[] coloredArray = new ColoredButton[5];
```

```

final JPanel southPanel = new JPanel();
final JButton connectButton = new JButton("Connect");
final JButton disconnectButton = new JButton("Disconnect");
final JButton clearButton = new JButton("Clear");

public MyClientFrame() {

    super(TITOLO);

    MyClientFrameListener ascoltatore = new MyClientFrameListener(this);

    northPanel.add(startButton);
    northPanel.add(ipLabel);
    northPanel.add(ipTextField);
    northPanel.add(portLabel);
    northPanel.add(portTextField);
    northPanel.add(terromopiButton);

    startButton.addActionListener(ascoltatore);
    startButton.setActionCommand(MyClientFrameListener.START_BUTTON);
    terromopiButton.addActionListener(ascoltatore);
    terromopiButton.setActionCommand(MyClientFrameListener.INTERROMPI_BUTTON);

    for (int i = 0; i < coloredArray.length; i++) {

        coloredArray[i] = new ColoredButton(i+ "", Color.LIGHT_GRAY);

        centralPanel.add(coloredArray[i]);
    }

    southPanel.add(connectButton);
    southPanel.add(disconnectButton);
    southPanel.add(clearButton);

    connectButton.addActionListener(ascoltatore);
    connectButton.setActionCommand(MyClientFrameListener.CONNECT_BUTTON);
    disconnectButton.addActionListener(ascoltatore);
    disconnectButton.setActionCommand(MyClientFrameListener.DISCONNECT_BUTTON);
    clearButton.addActionListener(ascoltatore);
    clearButton.setActionCommand(MyClientFrameListener.CLEAR_BUTTON);

    this.getContentPane().add(northPanel, BorderLayout.NORTH);
    this.getContentPane().add(centralPanel, BorderLayout.CENTER);
    this.getContentPane().add(southPanel, BorderLayout.SOUTH);

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(LARGHEZZA, ALTEZZA);
    this.setLocationRelativeTo(null); //piazza la finestra al centro
    this.setVisible(true);
}

-----

```

```

package client;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;
import java.util.StringTokenizer;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JOptionPane;

```

```

public class MyClientFrameListener implements ActionListener {

    MyClientFrame riferimentoFinestra;

    Socket socket; //li metto come variabili d'istanza perch oltre a istanziarle devo
                    chiuderle
    Scanner sc;
    PrintWriter pw;

    static boolean erroreDiConnessione = false;

    static final String START_BUTTON = "start";
    static final String INTERROMPI_BUTTON = "interrompi";
    static final String CONNECT_BUTTON = "connect";
    static final String DISCONNECT_BUTTON = "disconnect";
    static final String CLEAR_BUTTON = "clear";

    public MyClientFrameListener(MyClientFrame riferimentoFinestra) {

        this.riferimentoFinestra = riferimentoFinestra;
        this.setButtons(false, false);
    }

    @Override
    public void actionPerformed(ActionEvent e) {

        if (e.getActionCommand().equals(START_BUTTON)) {

            boolean valido = true;

            for (int i = 0; i < riferimentoFinestra.coloredArray.length; i++) {

                if (riferimentoFinestra.coloredArray[i].getDigit() == null ||
                    riferimentoFinestra.coloredArray[i].getDigit().equals("")) {

                    JOptionPane.showMessageDialog(null, "Il bottone n" + i + " non ha un
                        numero inserito");

                    valido = false;
                }
            }
        }
    }
}

```

```

        }

    }

    if (valido == true) {

        //disable clicks on the colored buttons

        this.setButtons(true, true);

        pw.println("start");
        pw.flush();

        //per gestire i dati che entrano dal server, creiamo un thread
        //quindi facciamo qualcosa che nel corso non abbiamo ancora visto
        //per vabbe lo creiamo in un altro file. Un'applicazione con pi threads
        //pu fare pi cose contemporaneamente. Ci serve perch vogliamo poter
        //usare l'interfaccia grafica mentre riceviamo dati, cio usare le due
        //cose contemporaneamente. SENZA QUESTO INFATTI NON SAREI IN GRADO DI
        //PREMERE IL TASTO INTERROMPI MENTRE MI ARRIVANO I DATI DA ELABORARE DAL
        //SERVER. Inoltre i dati ci vengono mandati ogni tot. secondi, quindi ci
        //conviene usare un thread che si occupa solo della lettura dei dati.
        //Quindi ora lo creiamo in un'altra classe, il codice lo capiremo pi
        //avanti.

        DownloaderThread dt = new
            DownloaderThread(riferimentoFinestra.coloredArray, sc, this);

        Thread t = new Thread(dt); //queste due opzioni servono per
        t.start();               //per avviare il thread
    }

}

else if (e.getActionCommand().equals(INTERROMPI_BUTTON)) {

    this.setButtons(true, false);

    pw.println("interrompi");
    pw.flush();

}

else if (e.getActionCommand().equals(CONNECT_BUTTON)) {

    this.setupConnection();

    if (!erroreDiConnessione) {
        this.setButtons(true, false);
    }

    erroreDiConnessione = false;
}

else if (e.getActionCommand().equals(DISCONNECT_BUTTON)) {

    this.closeConnection();

    this.setButtons(false, false);
}

```

```

    }

    else if (e.getActionCommand().equals(CLEAR_BUTTON)) {

    }

}

private void setupConnection() {

    try {

        socket = new Socket(riferimentoFinestra.ipTextField.getText(),
            Integer.parseInt(riferimentoFinestra.portTextField.getText()));

        pw = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
            //socket.getOutputStream() restituisce uno stream di Byte, quindi gli
            applichiamo un Writer

        sc = new Scanner(socket.getInputStream());

    } catch (IOException ex) {
        erroreDiConnessione = true;
        JOptionPane.showMessageDialog(null, "Impossibile connettersi al server");
    }
}

private void closeConnection() {
    sc.close();
    pw.close();
    try {
        socket.close();
    } catch (IOException ex) {
        JOptionPane.showMessageDialog(null, "Impossibile chiudere la connessione");
        Logger.getLogger(MyClientFrameListener.class.getName()).log(Level.SEVERE,
            null, ex);
    }
}

public void setButtons(boolean connesso, boolean start) { //facciamo un metodo
    apposito cos da pulire l'actionPerformed

    if (connesso && !start) { //you pressed Connect

        riferimentoFinestra.connectButton.setEnabled(false);
        riferimentoFinestra.interrompiButton.setEnabled(false);
        riferimentoFinestra.clearButton.setEnabled(true);
        riferimentoFinestra.startButton.setEnabled(true);
        riferimentoFinestra.disconnectButton.setEnabled(true);

    }

    if (connesso && start) { //you pressed Start

        riferimentoFinestra.connectButton.setEnabled(false);
        riferimentoFinestra.startButton.setEnabled(false);
    }
}

```

```

        riferimentoFinestra.disconnectButton.setEnabled(false);
        riferimentoFinestra.clearButton.setEnabled(false);
        riferimentoFinestra.interrompiButton.setEnabled(true);

        for (ColoredButton cb: riferimentoFinestra.coloredArray) {
            cb.changeColor(Color.LIGHT_GRAY);
        }
    }

    if (!connesso && !start) { //you pressed Disconnect or you never connect

        riferimentoFinestra.interrompiButton.setEnabled(false);
        riferimentoFinestra.startButton.setEnabled(false);
        riferimentoFinestra.disconnectButton.setEnabled(false);
        riferimentoFinestra.clearButton.setEnabled(true);
        riferimentoFinestra.connectButton.setEnabled(true);
    }

}

```

```

package client;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;

@SuppressWarnings("serial") //if we don't want to fix a warning we can suppress it like
    this.
public class ColoredButton extends JButton {

    //stringa che mantiene l'informazione sulla posizione della casella nella griglia
    private String position;
    //colore della casella
    private Color color;
    //cifra visualizzata all'interno della casella
    private String digit;

    public ColoredButton(String position, Color color) {
        this.position = position;
        this.color = color;
        this.digit = "";

        DigitListener listener = new DigitListener();
        this.addActionListener(listener);

        changeColor(color);
    }

    public void setTextDigit(String digit) {
        // metodo per gestire l'informazione riguardo la cifra visualizzata
        // all'interno della casella
        // passare come parametro la stringa vuota "" per ripristinare il contenuto di
        // una casella
        this.digit=digit;
        super.setText(digit);
    }

    public void changeColor(Color c) {

```

```

        // metodo per gestire l'informazione riguardo ai cambi di colore della casella
        this.color = c;
        super.setBackground(c);
    }

    public boolean isGreen() {
        // metodo che indica se la casella in questione contiene un numero vincente
        return color.equals(Color.GREEN);
    }

    public String getDigit() {
        //metodo utile per verificare a-posteriori se la casella in questione contiene
        //una cifra o una stringa vuota ""
        return digit;
    }

    @Override
    public void setBackground(Color bg) {
        // NON UTILIZZARE setBackground per cambiare colore ad un'istanza di
        // ColoredButton.
        // Usare invece il metodo changeColor(Color color)
        return;
    }

    @Override
    public void setText(String str) {
        // NON UTILIZZARE setText per cambiare il testo visualizzato ad un'istanza di
        // ColoredButton.
        // Usare invece il metodo setTextDigit(String n)
        return;
    }

}

final class DigitListener implements ActionListener {

    //listener per cambio cifra con il click
    protected DigitListener(){
        super();
    }

    public void actionPerformed(ActionEvent e) {
        ColoredButton cb = (ColoredButton) e.getSource();
        Reader r = new NumberInput();
        String digit = r.read();

        if (digit.equals("Cancel_Button")) {
            //don't do anything
        }

        else { // we pressed the ok button (and not the cancel button)
            cb.setTextDigit(""+digit);
            cb.changeColor(Color.LIGHT_GRAY);
        }
    }
}
-----*/
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

```

```

/*
package client;

import java.awt.Color;
import java.util.Scanner;
import javax.swing.JOptionPane;

/**
 *
 * @author studente
 */
public class DownloaderThread implements Runnable { //un Thread implementa sempre Runnable

    private ColoredButton[] pulsantiera;
    private Scanner sc;
    private MyClientFrameListener cl;

    public DownloaderThread(ColoredButton[] pulsantiera, Scanner sc,
                           MyClientFrameListener cl) {
        //bisogna passargli la lista dei pulsanti perch ci deve agire sopra
        //inoltre deve poter leggere con lo scanner e chiamare il Listener del Client

        this.pulsantiera = pulsantiera;
        this.sc = sc;
        this.cl = cl;
    }

    @Override
    public void run() {

        setEnabled(false); //dobbiamo disabilitare i pulsanti (non possiamo cambiare cosa
                          facciamo mentre riceviamo dati)

        boolean interrupted = false;

        boolean running = true;
        while(running) { //ora dobbiamo leggere riga per riga dal server.

            String cmd = sc.nextLine(); //leggiamo il comando mandatoci dal server

            //adesso per separare posizione e numeroEstratto dal comando ricevuto possiamo
            //usare un Tokenizer o il metodo split
            String commands[] = cmd.split(":");

            if (commands[0].equals("*") && commands[1].equals("*")) { //il server ha
                finito di inviare dati
                running = false;
                continue;
            }

            if (commands[0].equals("-1") && commands[1].equals("-1")) { //abbiamo premuto
                il tasto Interrompi, e il server ci ha risposto con -1:-1
                //in questo caso la partita l'ho persa, me lo devo segnare in qualche
                modo. Lo faccio usando una variabile interrupted
                interrupted = true;

                running = false;
                continue;
            }
        }
    }
}

```

```

        int posizione = Integer.parseInt(commands[0]);
        String numeroEstratto = commands[1];

        if (pulsantiera[posizione].getDigit().equals(numeroEstratto)) {
            pulsantiera[posizione].changeColor(Color.GREEN);
        } else {
            pulsantiera[posizione].changeColor(Color.RED);
        }
    }

    if (interrupted) {
        //visualizziamo il messaggio che dice che l'utente ha perso
        JOptionPane.showMessageDialog(null, "You Lose!!!");
    } else {
        //dobbiamo controllare se c' almeno un pulsante verde: solo in tal caso
        l'utente ha vinto

        boolean greenFound = false;
        for (ColoredButton cb: pulsantiera) {

            if (cb.isGreen()) {
                greenFound = true;
                break;
            }
        }

        if (greenFound) {
            JOptionPane.showMessageDialog(null, "You Win");
        } else {
            JOptionPane.showMessageDialog(null, "You Lose");
        }

        setEnabled(true); //finita la partita dobbiamo riabilitare la possibilit di
        cliccare sui pulsanti

        cl.setButtons(true, false);
    }
}

private void setEnabled(boolean state) { //prende in input lo stato dei pulsanti

    for (ColoredButton cb: pulsantiera) {

        cb.setEnabled(state);
    }
}

-----
package client;

public interface Reader {
    String read();
}

-----
package client;

```

```

import javax.swing.JOptionPane;

public class NumberInput implements Reader {

    @Override
    public String read() { //se non vogliamo che ci siano due numeri uguali
        //nelle caselle possiamo passare al read() tutti i numeri
        //inseriti attualmente nelle caselle, cos poi da non farli
        //reinserire all'utente

        // Questo metodo ed il return devono essere modificati a cura dello studente
        // in maniera da acquisire in input una cifra compresa tra 0 e 9

        String numeroStringa = JOptionPane.showInputDialog(null, "Inserire il numero:
        ");

        if (numeroStringa == null) { //you pressed the Cancel button on the Dialog (so
            if you press Cancel showInputDialog() returns null
            return "Cancel_Button";
        }

        boolean correctDigit = false; //cos gestiamo l'inserimento di stringhe che non
        sono numeri nella casella

        while (!correctDigit) {

            try {

                while (Integer.parseInt(numeroStringa) < 0 ||
                    Integer.parseInt(numeroStringa) > 9) {

                    JOptionPane.showMessageDialog(null, "Il numero inserito non
                    compreso tra 0 e 9. Riprovare: ");

                    numeroStringa = JOptionPane.showInputDialog(null, "Inserire il
                    numero: ");

                    if (numeroStringa == null) { //you pressed the Cancel button on the
                        Dialog (so if you press Cancel showInputDialog() returns null
                        return "Cancel_Button";
                    }
                }

                correctDigit = true;
            } catch (NumberFormatException e) {
                numeroStringa = JOptionPane.showInputDialog(null, "Inserire il numero:
                ");
            }
        }

        return Integer.parseInt(numeroStringa) + "";
    }

}

-----
package client;

```

```

public class Main {

    public static void main(String[] args) {

        MyClientFrame finestraUno = new MyClientFrame();

    }

}

```

42.3 Classe InetAddress:

Ogni istanza di questa classe rappresenta un indirizzo IP (sia in versione 4 che in versione 6). Per creare un oggetto di questa classe possiamo fare in una di queste maniere:

- static InetAddress[] getAllByName(String host); prende in ingresso una stringa, cioè il nome simbolico dell'ip, e restituisce un array di oggetti. Questo perché il DNS potrebbe ritornare diversi IP, non solo uno, a partire da un nome simbolico (es. google.com).
- static InetAddress getByAddress(byte[] addr); costruttore (non puoi istanziare direttamente getByAddress). Prende un array di byte, se l'indirizzo è v4 sono 32 bit, se v6 è 32 byte.
- static InetAddress getByAddress(String host, byte[] addr); prende in ingresso oltre all'indirizzo anche il nome dell'host. Questo, a differenza del getAllByName, non fa ricorso al DNS, e quindi fa creare indirizzi IP a piacere, anche se non esistono.
- static InetAddress getByName(String host); uguale a getAllByName ma restituisce solo il primo indirizzo IP disponibile
- static InetAddress getLocalHost(); restituisce il local host (quindi 127.0.0.1 nella maggior parte dei casi)

I metodi interessanti da chiamare su un'istanza di INetAddress sono:

- String getHostSAddress(); fa una pretty print dell'indirizzo e del nome del server
- String getHostName(); dall'indirizzo IP si può risalire al nome della macchina
- boolean isReachable(int timeout); effettua un ping su una macchina, un ping è una sorta di messaggio isAlive (cioè sto chiedendo alla macchina "ci sei?" e lei risponde). Quindi ci dice se la macchina c'è e quanto ci mette a rispondere. In realtà questo è un abuso di notazione, poiché in realtà il ping è solo il comando che l'utente dà alla macchina per capire se c'è e con quanta latenza ci risponde. Il termine proprio e corretto per la latenza stessa sarebbe gitter.

POI ci sono una serie di slide che mostrano come servire più client contemporaneamente, ma le vediamo dopo, dopo aver visto il thread. Vediamo però l'ultima slide, legata a come chiudere le connessioni.

42.3.1 Chiudere le Connessioni:

Dovremmo sempre cercare di fare pulizia, chiamando il metodo close() una volta che non ci servono più.

- nel caso di Serversocket close() termina la accept. Nel caso di monothread questa cosa non ha senso, vedremo invece come è utile per il multithread
- nel caso di socket la close() termina le operazioni di lettura/scrittura

Nota che sia Socket che ServerSocket hanno un metodo isClosed() che restituisce true se il socket è stato chiuso.

43 Thread, concetti di base:

Oggi siamo abituati a usare più app contemporaneamente, ma in realtà il concetto di contemporaneo nell'informatica non esiste: il processore fa al massimo un'istruzione alla volta, anche se è vero che con più core posso fare più istruzioni alla volta. Il discorso è però che i core condividono parte dell'hardware col processore, quindi in alcuni casi uno si deve mettere in attesa dell'altro. Perciò quattro core non significano automaticamente quattro istruzioni alla volta. Perché allora noi ci sembra di fare più cose contemporaneamente? Succede che l'OS permette ai processori di passare la palla in continuazione ai processi, quindi passa da un processo all'altro ogni 16 nanosecondi (per windows). Passare la palla significa che per 16 nanosecondi fa un processo, poi cambia a un altro. La nostra sensazione è quella di contemporaneità delle esecuzioni.

Monotasking: Questa cosa nei primi sistemi informatici non esisteva, quindi i sistemi erano mono-tasking: quando finisce un programma ne parte un altro, punto. Il problema principale del monotasking è che si spreca tempo di CPU. Infatti, dato che esistono processi CPU bound (il processo sfrutta principalmente la CPU) e IO bound (sfrutta principalmente Input/Output), e dato che l'IO è tantissimo più lento rispetto alla CPU, molto spesso il processore nel monotasking rimane a non fare niente perché sta aspettando che termini un processo IO bound, cioè legato all'input/output.

Multitasking: Quindi siamo passati al multitasking, in cui i processi differenti potevano essere eseguiti "contemporaneamente". Il problema del multitasking è che il passaggio da un processo all'altro, cioè il cambio di contesto (ogni 16 nanosecondi), è pesante e poco efficiente.

Multithreading: Quindi siamo andati oltre il multitasking, con il multithreading: ogni singolo processo è diviso in più sottoprocessi, i thread appunto. Un thread è un flusso di esecuzione indipendente in un programma. Nella nostra macchina contemporaneamente abbiamo migliaia di thread che girano in continuazione. Il vantaggio rispetto al multitasking è che passare la palla da un thread all'altro è più facile e leggero. L'OS prende un processo e lo esegue, cambiando ogni tot. nanosecondi il thread, come prima faceva coi processi. Questo è meglio del multitasking perché cambiare thread ogni tot. secondi costa meno ed è più efficiente di cambiare processo. Quindi la differenza tra il multitasking e il multithreading è che nel multitasking ad ogni programma corrisponde uno e un solo thread, il thread in cui gira il main, mentre nel multithreading ho più thread per ogni processo. Ma a cosa mi potrebbero servire i flussi di esecuzione? Ad esempio per quello che abbiamo fatto nell'esercizio del lotto: io voglio poter leggere da input ed elaborare i dati che mi vengono dal server (lo faccio tramite la classe DownloaderThread) e intanto poter continuare a utilizzare l'interfaccia grafica (che ho costruito nelle classi MyClientFrame e MyClientFrameListener), così da poter cliccare il tasto Interrompi. Nel monothread dovrei fare delle chiamate non bloccanti, cioè un metodo nextLine() che se non ci sta nulla da legge ritorna immediatamente """. QUindi in continuazione leggeva messaggi e intanto era utilizzabile l'interfaccia. Ma comunque meglio utilizzare il multithreading. Come viene scelto il thread da far girare ogni momento dipende dall'OS⁴⁶. Un altro modo in cui vengono chiamati i thread è lightweight process, proprio perché il cambio di processo ogni 16 nanosecondi è più leggero. Questo perché non devo cambiare area di memoria quando cambio thread, come invece succedeva cambiando processo.

43.1 Condivisione di Memoria nei thread:

Tutti i thread in un processo condividono la stessa area di memoria dinamica. Quindi il problema è che se ho più thread che accedono alla stessa memoria ho inconsistenza della memoria. Lo stack è indipendente, cioè è un'area specifica di ogni thread: se io chiamo la funzione da thread differenti, il contenuto delle var su quelle chiamate è indipendente da thread a thread.

43.2 Thread in Java:

In java abbiamo la classe `java.lang.Thread`. Le istanze di `Thread` sono un'interfaccia verso la JVM che è l'unica che può creare nuovi thread. Questa classe ci permette di creare nuovi thread,

⁴⁶per gli OS general purposes: Ad ogni thread è assegnata una priorità (ci sono thread più importanti). Es. anche se il sistema si impatta il mouse funziona: il processo del mouse è ad altissima priorità, per dare all'utente idea di continua interazione.

interromperli o attendere la fine di un thread. Semplicemente creando un istanza di Thread però non stiamo creando un Thread, perché dobbiamo anche usare il metodo start().

43.3 Implementazione dei Thread in Java, i due modi:

- Utilizziamo solo Thread (sconsigliato). Andiamo a creare una classe che estende Thread, e chiamiamo su questa classe il metodo run, facendo overriding. Si tratta del METODO NAIVE, non ci piace perché in Java se estendiamo Thread non possiamo estendere nient'altro, magari una classe più utile. Facciamo un esempio, ma ricordiamo che questo metodo è sconsigliato:

```
public class MioThread extends Thread {  
  
    private String nome;  
    public MioThread(String n) {  
  
        // super();  
        nome = n;  
  
    }  
    public void run() {  
  
        for (int i = 0; i < 10; i++) {  
  
            System.out.println(nome + ": " + i);  
  
        }  
  
        System.out.println(nome + ": DONE!");  
  
    }  
  
}  
  
-----  
  
public class Main {  
  
    public static void main(String[] args) {  
  
        MioThread jat = new MioThread("Jamaica");  
        jat.start();  
  
    }  
}
```

Il nostro thread estende Thread (quindi già capiamo che questo metodo è possibile solo nel caso di classi non derivate) e quindi deve fare overriding del run() di Thread. Attenzione che è un altro run() rispetto a quello di Runnable: nel primo caso facciamo overriding di run() di Runnable, mentre qui stiamo facendo run() di Thread (Thread non è una classe astratta, il suo run() non fa nulla). Poi per avviare un nuovo thread creo un'istanza di mio thread e su questa istanza chiamo start.

- utilizzo interfaccia Runnable (consigliato): con l'esercizio del lotto abbiamo usato questo. Passiamo un'istanza della classe che implementa Runnable al Thread e dal Thread chiamiamo start. In questo modo resta libero il meccanismo di estensione delle classi. Questo è un design pattern classico dell'ingegneria informatica: il pattern executor. L'oggetto che implementa l'interfaccia Runnable funge da eseguibile (cioè contenitore dell'istruzione da eseguire, infatti l'operazione da eseguire è dentro il metodo run() che viene costruito nella classe che implementa Runnable), mentre il Thread funge da esecutore: noi passiamo al Thread un'istanza della classe che implementa Runnable, e poi il thread usa su questa il metodo start() per eseguire l'istruzione in run(). Start() precisamente crea un nuovo thread ed esegue il metodo

run che abbiamo implementato in questo thread. Il metodo start() ha come oggetto di invocazione un'istanza di Thread, e crea il thread corrispondente all'oggetto di invocazione. Inoltre start invoca sul nuovo thread il metodo run() dell'oggetto della classe che implementa Runnable associato al Thread. Nota bene che è possibile invocare direttamente (cioè fuori da start()) il metodo run(), ma in questo caso non viene avviato un nuovo thread ma il metodo run() viene eseguito nel thread corrente, cioè quindi stiamo facendo come abbiamo sempre fatto per l'invocazione di un metodo in un programma: lo stiamo semplicemente chiamando per usarlo nel programma. Facciamo un esempio di questo metodo, che ricordiamo essere quello da preferire:

```

public class MioRunnable implements Runnable {

    private String nome;

    public MioRunnable(String n) {
        nome = n;
    }

    public void run() {

        for (int i = 0; i < 10; i++) {
            System.out.println(nome + ": " + i);
        }

        System.out.println(nome + ": DONE! ");
    }
}

-----
public class Main {

    public static void main(String[] args) {
        MioRunnable ja = new MioRunnable("Jamaica");
        Thread mioThread = new Thread(ja);
        mioThread.start();
    }
}

```

Quindi che facciamo: creiamo la classe MioRunnable che implementa Runnable. Essa ha un costruttore (che prende in ingresso una stringa che è il nome del thread) e il metodo run() (ciclo di 10 volte che stampa il nome e il numero d'iterazione). Non appena termina l'esecuzione del metodo run() il thread muore, così come quando il main finisce in un programma muore il programma. Vediamo nel main che prima creiamo il runnable e poi il thread, e SOLO quando chiamiamo MioThread.start() l'applicazione si biforca e ho due thread, che girano. In questo caso manco ce ne accorgiamo perché l'esecuzione è rapida: chiamiamo mioThread.start(), subito dopo muore il main e poi stampe velocissime e muore anche il thread nuovo e termina l'applicazione.

Dato che ci siamo, vediamo dei dettagli sulla classe Thread:

- ha due costruttori, il primo è vuoto e viene sfruttato nella soluzione naïve, l'altro è il costruttore che prende in ingresso un'istanza di runnable, che utilizziamo nella soluzione consigliata.
- inoltre come detto ha un metodo run() che non fa nulla. Se passiamo un'istanza di runnable il metodo start() chiama il metodo run() dell'istanza di Runnable, se invece non passiamo questa istanza il metodo start() chiama proprio questo run(), il run() della classe thread, di cui dobbiamo quindi fare overriding nella nostra classe che estende Thread, anche perché il run() di Thread non fa nulla, come già detto.

Poi abbiamo l'interfaccia Runnable, che come abbiamo detto è composta solo dal metodo run().

43.4 Esercizio:

Realizzare una classe Java Runnable che rappresenta un eseguibile per stampare (senza andare a capo) su un output 100 volte un carattere passato come parametro.

- Scrivere un main in cui vengono creati e avviati due thread, il primo che stampa il carattere '0', il secondo il carattere '1'
- Eseguire più volte il programma e verificare se le sequenze di 0 e 1 generate sono uguali

```
public class PrintExecutable implements Runnable {

    private final char c;
    public PrintExecutable(char c) {
        this.c = c;
    }

    public void run() {
        for (int i=0;i<100;i++)
            System.out.print(c);
    }
}

public class Main {

    public static void main(String[] args) {
        Thread t1 = new Thread(new PrintExecutable('0'));
        Thread t2 = new Thread(new PrintExecutable('1'));
        t1.start();
        t2.start();
    }
}
```

I thread sono eseguiti parallelamente infatti in console potremmo, ad esempio, avere:

```
00000000000011111111111111110000000000000011111111111
```

Dove la lunghezza di una riga di 0 rappresenta il *quanto* che il processore ha dedicato al thread che stampa 0, idem per ogni riga di 1. Finito il tempo di quel thread il processore passa la palla all'altro thread, e così via. Può succedere che l'OS fa preemption (pre-rilascio), cioè l'OS si accorge che deve fare una cosa fondamentale e toglie tempo al thread, quindi per questo non tutte le righe sono di egual lunghezza. Se andiamo a vedere nelle slide, c'è una stampa di prima tutti 0 e poi tutti 1; questo perché a seconda di quanto è veloce il processore quello che può succedere è che nel tempo che io ci metto ad avviare il secondo thread il primo ha già finito, quindi dipende da come è fatto il sistema e dalle prestazioni. C'è però un modo per forzare in un ballo più continuo i thread: utilizzando il metodo sleep(). Il metodo sleep() del thread mette a dormire il thread, lo mette quindi in pausa. Questo metodo prende in ingresso un long (intero a 64 bit) che è il numero di millisecondi in cui ci vogliamo mettere in attesa. Ogni thread è una macchina a stati, ogni momento è in uno stato, sleep() porta dallo stato runnable (in cui il thread accetta tempo cpu e fa qualcosa) allo stato non runnable (v. dopo). Attenzione che sleep() lo dobbiamo intendere come tempo per il quale il thread è in non runnable, quando il tempo scade, non è detto che l'OS poi abbia voglia di farlo rigirare subito. Nota anche che thread.sleep() lancia un'eccezione checked da gestire di tipo InterruptedException (che poi vediamo).

```
public class PrintExecutable implements Runnable {

    private final char c;

    public PrintExecutable(char c) {
        this.c = c;
    }

    public void run() {
```

```

        for (int i=0;i<100;i++){
            System.out.print(c);

            try {
                Thread.sleep((long)(Math.random() * 10));
            } catch (InterruptedException e) {}
        }
    }

-----
public class Main {

    public static void main(String[] args) {
        Thread t1 = new Thread(new PrintExecutable('0'));
        Thread t2 = new Thread(new PrintExecutable('1'));
        t1.start();
        t2.start();
    }
}

```

quindi con sleep() il thread sta dicendo all'OS "ok basta, finisco qua il mio quanto, e non me ne dare altro di tempo minimo fino a Math.random()*10 millisecondi passati, poi da lì poi ridarmelo se vuoi. Se eseguiamo in questo caso i numeri sono più intervallati, esempio

01010101011010000101101010101110010111

però aspetta, perché ho degli 0 e degli 1 insieme? Ho messo degli sleep quindi dovrei fare sempre 10101010101010101. In realtà può essere che gli intervalli di sleep dei due thread si sovrappongano e che quando escono entrambi dallo sleep l'OS decide di dare il tempo a quello a cui l'aveva dato prima ad esempio. Quando un thread non riesce a ricevere Tempo CPU il thread si dice che va in starvation. C'è un altro modo per fare la stessa cosa, ed è il metodo yield(). thread.yield() dice guarda io non voglio più usare il mio quanto, passalo a qualcun altro. yield() è quindi identico a passare 0 a sleep()

43.5 Variabili nei Thread:

Utilizzo variabili nei thread: qui dobbiamo fare una distinzione, abbiamo visto fin'ora le var locali all'esecuzione di un metodo (definite nello stack). Abbiamo detto che lo stack è indipendente, ogni stack ha il suo. Quindi le var locali di un metodo in un thread sono indipendenti dagli altri thread e propri solo di quel thread. Attenzione parliamo delle variabili non degli oggetti a cui quelle variabili puntano!!! Come sappiamo gli oggetti invece vengono creati nell'heap, memoria dinamica, che è invece condivisa tra più thread. Quindi può succedere che più thread puntino, con variabili separate e indipendenti, allo stesso oggetto in memoria (perché questo è nell'heap). Le variabili di istanza invece sono comuni a tutti i thread, perché stiamo accedendo allo stesso oggetto, che sta nell'heap. Anche le var statiche di classe sono comuni a tutti i thread, unica cosa specifica di un thread e indipendente dagli altri sono quindi le variabili locali di un metodo, ma solo le variabili, non gli oggetti, il contenuto, a cui essi puntano! Cioè se io ho due thread che accedono allo stesso oggetto in memoria devo stare attento, perché possono succedere cose strane, quindi occorrono metodi per sincronizzare più thread. Quindi, per capire:

```

public class Test {

    static int staticVAR; //comune a tutti i thread che fanno Test.staticVAR
    Object instanceVAR; //comune a tutti i thread che fanno istanzaDiTest.instanceVAR se
                       //tutti i thread stanno usando la stessa istanza di Test

    void methodFOO(int paramP) {

        int localVAR = paramP; //sono specifiche di ogni thread e solo sue
    }
}

```

```

    ...
}

}

```

e volendo fare un esempio:

```

public class RunExecutable implements Runnable {
    public void run() {
        for (int i = 0; i < 25; i++) {
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " is running");
        }
    }
}

-----
public class Main {
    public static void main(String[] args) {

        System.out.println(Thread.currentThread().getName() + " is running");

        Thread alpha = new Thread(new RunExecutable());
        Thread beta = new Thread(new RunExecutable());

        alpha.setName("Alpha thread");
        beta.setName("Beta thread");

        alpha.start();
        beta.start();

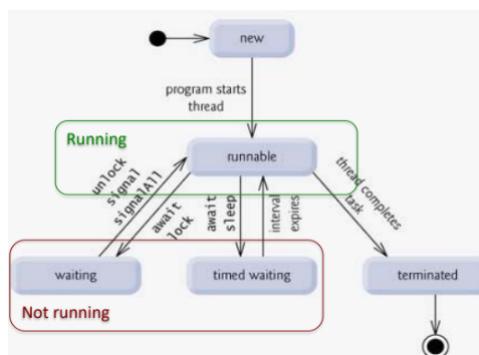
        System.out.println(Thread.currentThread().getName() + " stops running");

    }
}

```

Quindi abbiamo. Un metodo `setName()`, che dà un nome simbolico al thread. Un metodo `currentThread()`, metodo statico di `Thread` che fornisce l'istanza del thread che sta facendo girare il codice. Quindi in pratica sto creando due thread, gli sto dando in input due istanze diverse di `RunExecutable`, che implementa `Runnable`, e li sto chiamando `alpha` e `beta`. Quando li starto, su console verranno scritti "alpha is running" e "beta is running" in continuazione, finché non terminiamo. Vediamo che l'ultima istruzione viene eseguita subito dopo, perché non dobbiamo aspettare che finiscono i due thread che abbiamo startato, ma il thread del main continua in "contemporanea". Tra l'altro il `getName()` sul thread del main ci ridà "main", che è quindi il nome del thread iniziale, quello che parte dal main. Nota che ogni thread ha anche un ID, lo stesso usato dall'OS, ma non l'abbiamo usato.

43.6 Ciclo di Vita del Thread:



Questa è una macchina a stati, ed è quello che avviene dal punto di vista dell'OS proprio, non in Java. Precisamente abbiamo i seguenti stati:

- Stato new: un thread è in questo stato quando è stato creato ma non è ancora eseguibile
- Stato running: un thread è in questo stato quando può ricevere tempo CPU per fare l'operazione che deve fare
- Stato not running:
 - Stato waiting: avviene quando un thread attende che altri thread facciano qualcosa d'altro: è il meccanismo di sincronizzazione di prima. Cioè dico al thread "guarda aspetta che questo altro thread finisca di fare questa cosa e poi rientri in runnable". Da waiting si esce quando il thread che stavamo aspettando chiama signal, signalall o rilascia il lock su un determinato oggetto (c'è un lock quando un thread dice "guarda io devo lavorare su questo oggetto e ci devo lavorare solo io [es. magari è instabile e lo può gestire solo un thread], quindi chiunque altro deve lavorarci deve aspettare che io gli dica che ho finito di lavorarci sopra")
 - Stato timed waiting: avviene quando chiamo la sleep(), quando l'intervallo di sleep scade ritorno allo stato runnable.
- Stato terminated: quando un thread è completo, cioè l'esecuzione del run() termina, si va in terminated

Quindi in Java:

- Quando facciamo:

```
Thread t1 = new Thread(new PrintExecutable('0'));
```

il thread è in stato new.

- Quando chiamo start() il thread passa in stato runnable
- Quando chiamo sleep() il thread va in timed waiting, quando finisce l'intervallo della sleep() torna in runnable
- Se chiedo un input da console (ad esempio con (new Scanner(System.in)).nextLine()) in un thread quel thread va in waiting
- Quando il run() termina il thread va in stato terminated

Nota che posso verificare se un thread è terminato o meno chiamando il metodo isAlive(), che restituisce true se il thread sta in running o not running, mentre restituisce false se il thread non è mai stato avviato con start() o è in terminated.

Part XI

Seconda Parte, l'analisi:

Lo sviluppo del SW, da dopo la raccolta dei requisiti, presenta queste tre fasi:

- Analisi
- Progettazione
- Programmazione

L'analisi la vediamo adesso in profondità. La progettazione e la programmazione si occupano di tradurre in SW lo schema concettuale prodotto dall'analisi. Notiamo che sono finiti i tempi di Fondamenti di Informatica I o Tecniche di Programmazione dove si programmava "naive" dall'inizio, da ora useremo solo pattern di programmazione: ci studiamo come si risolve una certa cosa lo facciamo sempre così. La fase dell'analisi consiste in questo: dopo aver raccolto i requisiti dai committenti e dagli esperti del dominio, si vogliono formalizzare questi requisiti, cioè li si vuole esplicitare in maniera chiara (eliminando ogni ambiguità) e rigorosa in modo che, quando si dovrà programmare, i programmatore potranno tradurre in programma lo schema concettuale prodotto dall'analisi senza andare a riguardare i requisiti. Quindi l'analisi è la fase che ha come input i requisiti raccolti e come output uno **schema concettuale** (o modello d'analisi) dell'applicazione. Quindi l'analisi si occupa della concettualizzazione, cioè il modello matematico di quello che il SW deve fare. Attenzione: l'analisi dice COSA il SW dovrà fare, non COME. Questo significa che quando si fa l'analisi non si è ancora decisa la tecnologia con cui realizzare il SW⁴⁷ (cioè il COME). Ok quindi l'analisi produce uno schema concettuale, ma come? Non si lavora matematicamente con le formule ma si usano degli strumenti informatici per fare quest'analisi: i **diagrammi**, che sono come i blueprints dei palazzi per gli architetti. Lo schema concettuale è quindi composto da:

- Diagramma delle classi e degli oggetti (diagramma strutturale): descrive le classi dell'applicazione e i rapporti che hanno tra di loro. Inoltre descrive particolari oggetti significativi.
- Diagramma delle attività (uno dei diagrammi comportamentali): descrive le funzionalità che il sistema deve realizzare
- Diagramma degli stati e delle transizioni (uno dei diagrammi comportamentali): descrive, di una certa classe, il ciclo di vita delle sue istanze
- Documenti di specifica: descrivono quali condizioni i programmi che realizzeranno lo schema concettuale devono rispettare. Ogni classe ha il suo documento di specifica. Usiamo i documenti di specifica perché non riusciamo a scrivere tutto sui diagrammi, soprattutto per i diagrammi delle attività e i diagrammi degli stati e delle transizioni, mentre i diagrammi delle classi e degli oggetti sono abbastanza esaustivi.
- Altri diagrammi su cui non ci soffermeremo

In passato si faceva analisi orientata alle funzioni, quindi si costruivano diagrammi funzionali e di flusso di dati. Oggi invece si fa analisi orientata agli oggetti, basata sul linguaggio UML⁴⁸. Il fatto che l'analisi oggi sia orientata agli oggetti significa che prima si guarda quali sono gli oggetti (i dati) in gioco e poi si va a capire come manipolare queste informazioni, i diagrammi rispecchieranno quindi questa scelta. Infine diciamo che ci concentreremo sul progettare Applicazioni **Concorrenti**: cioè con più processi allo stesso tempo, coordinati tra di loro. In UML tutto è PUBLIC, ciò che è PRIVATE non interessa all'utente (quindi non interessa all'analista e al progettista ma solo al programmatore perché la roba public è funzionale al suo lavoro.)

⁴⁷Anche se noi utilizzeremo sempre Java, portandoci quindi dietro le peculiarità di questo linguaggio.

⁴⁸Unified Modelling Language. Nato dalla necessità di dotarsi di strumenti per l'analisi che risultassero comodi, in modo da non dover scrivere centinaia di migliaia di pagine di codice con formule matematiche. Quindi facendo analisi con UML si prendono decisioni ad alto livello piuttosto che scrivere ogni riga di codice. Si chiama così perché Jacobson, Rumbaugh e Booch si misero assieme e unificarono i loro tre sistemi proprietari di modellazione (per questo unified), inoltre UML è Open Source.

44 Diagramma delle attività:

Per rappresentare questo dinamismo delle entità, che cambiano STATO ma restano loro stesse, si utilizza questo diagramma. Attenzione che le attività non sono semplicemente delle funzioni: non parlano solo dell'input e dell'output, ma colgono cosa succede durante l'esecuzione dell'attività, i passi intermedi, le attività complesse con cui sequenziamo le attività atomiche tra l'input e l'output.

45 Diagramma degli Stati e delle Transizioni:

Diamo all'oggetto una vita, cioè gli diamo degli eventi e lui coglierà gli eventi e in base a che punto della sua vita sta coglie alcuni eventi o altri. Facciamo un esempio entrando nel mondo dei videogiochi: gli NPC sono i cattivi e tu je spari, ma vuoi che loro si comportino in modo interessante, che ne so se gli spari li ferisci e cercano di nascondersi. Quindi vuoi che loro cambino il loro comportamento. Cioè hanno uno stato e lo possono cambiare a fronte di eventi ricevuti. Quindi in un SW complesso ci sarà un sistema di persone: uno lancerà un evento, un'altro lo coglie e lo rilancia, l'altro non lo coglie ecc. . Questo significa che il nostro SW sarà un sistema che girerà e si muoverà sulla base degli eventi, che sono il suo motore.

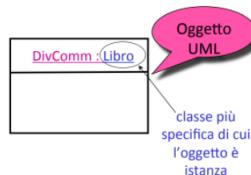
46 Diagrammi delle classi e degli oggetti:

Si tratta del più importante diagramma. Consideriamo che nella fase di analisi ci si concentra sulle classi più che sugli oggetti, ma il Diagramma delle classi e degli oggetti a volte può concentrarsi anche sugli oggetti se questi sono particolari elementi significativi, come già detto. Inoltre c'è da dire che **nell'analisi si lavora sull'astrazione di entità e non sull'astrazione di valori**. Se ci servirà rappresentare un'astrazione di valore, useremo i **tipi di dato astratti**: queste rappresentazioni, già viste, sono infatti astrazioni di valori, poiché non hanno l'elemento essenziale dell'entità, cioè lo **STATO**. Per spiegare questo concetto facciamo un esempio: un'entità Persona, rimane una Persona anche se cambia stato, cioè se da felice diventa triste, mentre il valore Numero, se cambia è un altro numero. Quindi le **entità hanno la caratteristica di rimanere sé stesse anche se cambiano stato**. Il motivo per cui non usiamo le astrazioni di valore negli schemi concettuali (ma se proprio ci serve facciamo una rappresentazione di tipo astratto) è che UML è orientato agli oggetti, e inoltre che noi non viviamo nella realtà i valori (i numeri non esistono nella realtà, sono un concetto astratto, nella realtà esistono le entità), e noi quando sviluppiamo SW stiamo facendo un modello della realtà.

46.1 Rappresentazione di Classi e Oggetti in UML:

46.1.1 Gli oggetti:

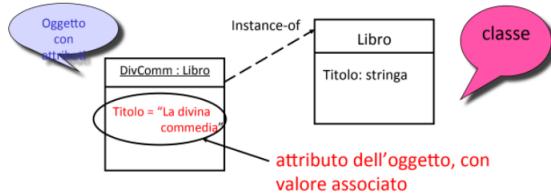
un oggetto in UML modella un elemento del dominio di analisi, identificato mediante l'identificatore. Un oggetto è istanza della sua classe più specifica. Gli oggetti formano il livello estensionale.



46.1.2 Le classi, gli attributi e Instance-Of:

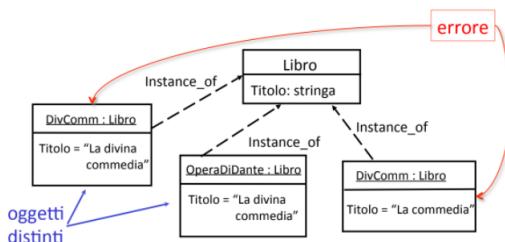
Le classi modellano un insieme di elementi (gli oggetti appunto). Ogni classe è descritta da un nome e un'insieme di proprietà locali (proprietà che tutti gli oggetti hanno in comune). Le classi formano il livello intensionale, che determina completamente il livello estensionale, poiché ne determina gli attributi. Un attributo modella una proprietà locale della classe, ed ha un nome seguito dal tipo del valore associato all'attributo. Gli attributi sono proprietà locali, cioè non possono avere come tipi dell'attributo degli oggetti, ma solo **dei valori** (es. intero, stringa ecc.). Questo ci porta alla distinzione tra oggetti e valori: un oggetto è istanza di una classe e **ha vita propria**, mentre un valore è un elemento di un tipo, quindi non ha vita propria ed **ha senso solo se associato a un**

oggetto tramite un attributo. Gli attributi sono i portali dal mondo degli oggetti al mondo dei valori. Possiamo pensare a un attributo di una classe come una Funzione che va dall'istanza della classe a dei valori. Cioè ha come dominio la classe e come codominio il tipo dell'attributo. In altre parole è una funzione che associa ad ogni oggetto istanziato in una classe un determinato valore che il tipo dell'attributo può assumere. Le funzioni sono totali rispetto al loro dominio: tutte le istanze di una classe devono avere un valore per quell'attributo (cioè il dominio è completo, es. TUTTI gli studenti devono avere un nome) e soltanto uno (è impossibile che uno studente abbia due nomi). Mentre il dominio è completo, non tutte le stringhe sono il nome di qualcuno, quindi il codominio non è completo. Importante: ogni istanza di una classe deve avere **tutti e soli**⁴⁹ i suoi attributi, dello stesso tipo deciso dalla classe. Vediamo che tra un oggetto e la sua classe di appartenenza



si traccia un arco Instance-of. Per tutto ciò che abbiamo detto finora, **due classi non possono avere istanze comuni:** classi diverse modellano insiemi disgiunti (v. dopo la *generalizzazione*).

L'importanza dell'identificatore di oggetto: due oggetti con identificatore diverso sono diversi, anche se hanno tutti gli attributi con gli stessi identici valori. Quando si fa la copia di un oggetto l'identificatore rimane lo stesso. Non si possono avere due oggetti con lo stesso identificatore ma con attributi diversi.



Associazioni (o relazioni) in UML: partiamo dalla definizione matematica di relazione: dati due insiemi S_1 e S_2 , una relazione R tra i due è un **sottoinsieme del prodotto cartesiano**⁵⁰ tra i due, cioè:

$$R \subseteq S_1 \times S_2$$

Quindi una relazione seleziona il sottoinsieme del prodotto cartesiano che è più significativo per R stesso. Ovviamente la nozione si può generalizzare al prodotto di n insiemi, e quindi alle relazioni tra n insiemi. Esempio:

$$S_1 = \{Paolo, Anna, Lucia\}$$

$$S_2 = \{Analisi, Geometria\}$$

Del prodotto cartesiano di questi due, che consta di tutte le possibili coppie di studenti e esami, prendiamo la relazione:

$$R = \text{Sostenuto} = \{(Paolo, Analisi), (Anna, Analisi), (Anna, Geometria)\}$$

⁴⁹UML permette solo oggetti poveri, non ricchi. Non si può quindi mettere informazione nell'istanza che non sia presente nella classe (es. non posso mettere l'attributo lingua nell'oggetto Iliade, istanza della classe Libro, se la classe libro non definisce questo attributo). Questa scelta permette di capire l'intera applicazione guardando le classi nel diagramma, poiché il livello intensionale definisce univocamente e completamente il livello estensionale. Se invece gli oggetti fossero stati ricchi avremmo dovuto guardare anche le specificità di ogni singolo oggetto, ponendo attenzione anche a loro. La possibilità di avere solo oggetti poveri viene adottata anche da Java.

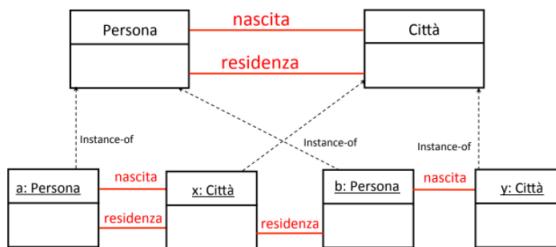
⁵⁰Il prodotto cartesiano tra due insiemi è l'insieme di tutte le coppie (x,y) t.c. x appartiene al primo insieme e y al secondo.

Notiamo che le relazioni permettono in generale⁵¹ che lo stesso oggetto del dominio vada in diversi oggetti del codominio, questo significa che un oggetto può avere una relazione con più oggetti. Le relazioni tra loro non sono collegate in nessun modo. Ok, adesso vediamo le relazioni in UML. Per il momento ci limitiamo a relazioni (chiamate anche associazioni) tra due classi, ma possiamo farlo anche tra n classi. **Una associazione tra due classi modella una relazione matematica tra i due insiemi delle istanze delle corrispettive classi.** Notare quindi la differenza tra attributi e associazioni: gli attributi modellano proprietà locali, cioè relative ad una e soltanto ad una classe, le associazioni modellano proprietà che coinvolgono due o più classi tra loro, quindi modellano una proprietà di tutte le classi coinvolte. Nessuna associazione è monodirezionale: studente è iscritto a corso e corso ha iscritto lo studente, posso navigare in entrambi i versi. Esempio di associazione: Come abbiamo detto le relazioni, tra loro non sono



collegate in nessun modo, ciò significa che se ho la classe di Studenti, e la relazione AMICI, questa non avrà nulla a che fare con la relazione FIDANZATI. Inoltre abbiamo visto matematicamente che un oggetto può avere una relazione con più oggetti, e questo vale anche in UML. Questo vale perché si tratta di relazioni, se si trattasse di funzioni non varrebbe. Infatti l'attributo che è una funzione non permette stessa cosa: a un oggetto (dominio) è associato uno e un solo valore specifico di un attributo (codominio), non posso associare a un oggetto più valori di un attributo.

Istanze di Associazione: come esistono le istanze di classe esistono anche le istanze di associazione. Queste istanze sono chiamate **link**, e collegano due oggetti. Cioè se A è associazione tra classi C_1 e C_2 , allora un'istanza di A è un link tra un oggetto di C_1 e uno di C_2 . Questo link è quindi, una n-pla: la n-pla composta dalle due istanze unite dal link (nel nostro caso, avendo visto solo associazioni tra due classi, si parlerà di coppie). Il significato del link è $(istanzaClasseC_1, istanzaClasseC_2) \in AssociazioneTraC_1eC_2$. Ecco un esempio, avendo fatto due associazioni tra le due classi Persona e Città, l'associazione nascita e l'associazione residenza: ATTENZIONE! L'associazione a livello intensionale tra due classi non dice nulla, a livello esten-



sionale, sul numero di link istanze dell'associazione in questione che coinvolgono due istanze delle due classi! Cioè un'istanza può essere linkata a 0, 1 o più istanze dell'altra classe, arbitrariamente. Successivamente vedremo come specificare condizioni sul numero di link. Esempio: se la classe Persona e la classe Città sono legati dall'associazione nascita, questo non ci dice nulla sul numero di istanze di Città al quale un'istanza di Persona è legata: un'istanza di Persona potrà avere 0,1 o più istanze di Città al quale è legata tramite dei link. Quando vedremo come specificare condizioni sul numero di link vedremo come imporre ad esempio che ogni istanza di Persona abbia esattamente un link di tipo nascita, cioè esattamente un'istanza di Città al quale è legata. Il discorso è che quindi:

$$iscritto \subseteq Studente \times CorsodiLaurea$$

mentre:

$$link \subseteq (Mario, Ingegneria) \in iscritto$$

⁵¹In generale perché un particolare tipo di relazione, la *funzione*, non permette ciò: una funzione manda un oggetto del dominio in al più un oggetto del codominio.

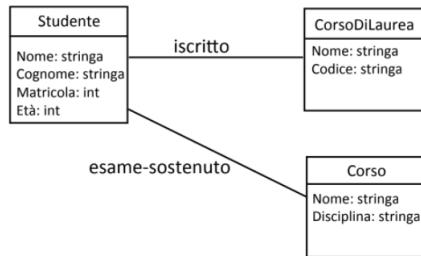
La n-pla è un'astrazione di valore: mentre gli object identifier servono per dare all'oggetto un'identità univoca e immutabile nel tempo (es. se Giulia da felice diventa triste sempre Giulia resta), la coppia è un'astrazione di valore, infatti non ha l'identificatore. Esempio: se prendo (Mario, Ingegneria) e cambio in (Mario, Architettura) non è cambiato lo stato ma è cambiata la coppia! Quindi le coppie non hanno stato, se cambi qualcosa cambi coppia, non sono entità!

46.1.3 Facciamo un Esempio:

Tracciare il diagramma delle classi corrispondenti alle seguenti specifiche: si vogliono modellare gli studenti (con nome, cognome, numero di matricola, età), il corso di laurea in cui sono iscritti, ed i corsi di cui hanno sostenuto l'esame. Di ogni corso di laurea interessa il codice e il nome. Di ogni corso interessa il nome e la disciplina a cui appartiene (ad esempio: matematica, fisica, informatica ecc.).

Quali sono i punti da seguire per risolvere un esercizio così? Eccoli:

- Estrai le classi: le classi sono le cose su cui si predica, cioè di cui si parla, su cui si dicono dei fatti, fatti che generalmente sono proprietà della classe, e cioè gli attributi (se riguardano solo e soltanto loro) o le associazioni (se riguardano anche un'altra classe).
- Estrai gli attributi: gli attributi devono parlare solo e soltanto di una classe (es. attributi di studenti parlano solo di studenti), e non possono esserci altre classi in mezzo. Questo perché gli attributi sono solo valori e non oggetti. Se pensiamo all'attributo come funzione che va da un dominio classe a un codominio tipo (interi stringhe ecc.), capiamo matematicamente perché non possono esserci in mezzo altre classi.
- Estrai le associazioni. Per trovarle vedi quelle proprietà che parlano di due (o più) classi. Se una proprietà è relativa a una singola classe è infatti un attributo, se invece parla di più classi è un'associazione



Osservazioni:

- Il fatto di essere iscritti a un corso non è un attributo perché è relativo a un'altra classe (appunto CorsoDiLaurea) e non riusciamo a codificarlo completamente solo come tipo stringa o intero, dato che gli attributi sono valori e non oggetti. Infatti qua il fatto di essere iscritti a un corso di laurea non può essere una funzione che va da un dominio classe a un codominio tipo (interi stringhe ecc.), perché c'è in mezzo un'altra classe: CorsoDiLaurea.
- Potremmo pensare che Disciplina debba essere una classe. Invece è una stringa, attributo di corso. Come mai? Perché di Disciplina non si predica: non abbiamo ulteriori informazioni che ci diano delle sue proprietà, da considerare come attributi. Quindi è un semplice attributo.
- Non modelliamo il fatto che i corsi siano contenuti in corsi di laurea perché non c'è scritto nel testo.

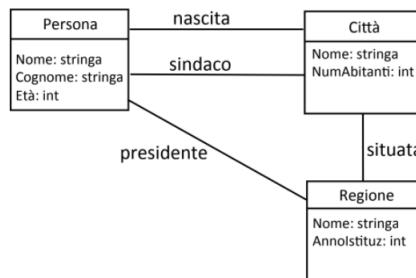
46.1.4 Altro esempio, differenza tra attributi e associazioni:

Tracciare il diagramma delle classi corrispondenti alle seguenti specifiche: si vogliono modellare le persone (con nome, cognome e età), le città di nascita (con nome, numero di abitanti e sindaco), e le regioni in cui si trovano le città (con nome, anno di istituzione e presidente).

Suggerimento: gli attributi non sono così importanti, ciò che caratterizza l'app sono classi e associazioni. Comodo è prima costruire classi, poi associazioni e poi le pile di attributi.

Risposta:

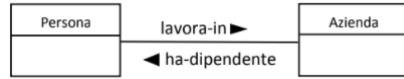
- Estrazione classi: vediamo che di persone, città di nascita e regioni si predica, poiché si parla di proprietà che hanno, che possono essere sia attributi (proprietà locali che hanno solo loro), sia associazioni (proprietà che condividono con un'altra classe tra le altre 2).
- Estrazione attributi:
 - Per Persona: nome, cognome e età sono proprietà di Persona che riguardano solo lei, non c'entrano nulla con le Città o le Regioni.
 - Per Città di Nascita: nome e numero di abitanti sono proprietà di Città di Nascita che riguardano solo lei; mentre sindaco non va bene: riguarda anche Persona.
 - Per Regione: nome e anno di istituzioni sono proprietà che riguardano solo lei; mentre presidente non va bene: riguarda anche Persona
- Estrazione associazioni:
 - Vediamo che "città di nascita" contiene un'informazione che lega due classi: Città di Nascita e Persona hanno come associazione nascita
 - Vediamo che la proprietà sindaco di Città di Nascita, che avevamo scartato come attributo poiché tirava in mezzo la classe Persona, è quindi un'associazione tra Persona e Città di Nascita
 - Vediamo che "regioni in cui sono situate" contiene un'informazione che lega due classi: Città di Nascita e Regione hanno come associazione situata
 - Vediamo che la proprietà presidente di Regione, che avevamo scartato come attributo poiché tirava in mezzo la classe Persona, è quindi un'associazione tra Persona e Regione



Errore Comune: prendere presidente e sindaco, che sono Persone, e metterle, in forma di Stringhe, come attributi rispettivamente di Regione e Città di Nascita. ERRORE!!! In questo modo si sta prendendo SOLO IL NOME del presidente o del sindaco, NON LA PERSONA intera. Quindi si sta prendendo un attributo della classe Persona, ma è sbagliato, bisogna rappresentare TUTTA la Persona, quindi sono associazioni, perché sono proprietà tra due classi. Questo è un errore concettuale, non sintattico (sintattico è mettere Persona come tipo di un attributo, poiché non si può proprio fare sintatticamente e UML ti ferma, mentre per l'errore concettuale UML non ti ferma).

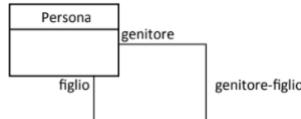
46.2 Versi e doppi nomi nelle Associazioni:

Si parla di zucchero sintattico: posso abbellire il diagramma con delle notazioni per chiarirlo. Esempio se ho la relazione nata-a tra Persona e Città posso mettere la freccetta che va da Persona a Città. Comunque la navigazione NON diventa monodirezionale, si naviga in ogni verso sempre, ma questa freccia ci dà un'idea di cosa sta dicendo il diagramma, il che può essere utile in diagrammi molto complicati. Si possono mettere anche due nomi: uno per la navigazione verso destra e uno per la navigazione verso sinistra. Frecce e doppi nomi si aggiungono sia nelle associazioni che nei link tra le istanze.



46.3 Ruoli nelle associazioni:

Invece di parlare di prima componenente e seconda componenente, si possono specificare i ruoli di queste componenti: es. se l'associazione è lavora-in, tra Persona e Azienda, posso chiamare dipendente la Persona e datore di lavoro l'azienda. Analogamente alle frecce e ai doppi nomi, i ruoli si aggiungono sia nelle associazioni che nei link tra le istanze. I ruoli però, differentemente da versi e doppi nomi, non sono sempre zucchero sintattico: c'è un caso in cui il ruolo è OBBLIGATORIO. Si tratta del caso in cui **l'associazione insiste più volte sulla stessa classe**, e inoltre **non rappresenta una relazione simmetrica**. Esempio: Se non avessimo i due ruoli "genitore"



e "figlio" posizionati, non sapremmo interpretare correttamente il link istanza dell'associazione "genitore-figlio". Infatti:

$$\text{genitore} - \text{figlio} \subseteq \text{Persona} \times \text{Persona}$$

prendiamo la coppia (giovanni, maria) \in genitore-figlio. Ok, ma chi è il genitore? Chi è il figlio? Potremmo decidere di ordinare in un certo modo la coppia, quindi dire "metto sempre prima genitore e poi figlio", ad esempio. Ma ancora meglio, così da non dover vincolare a precise posizioni gli elementi della coppia, **possiamo costruire una coppia etichettata scrivendo il nome del ruolo prima del nome della Persona, separandolo da quest'ultimo con ":".** Quindi, nel nostro caso:

$$<\text{genitore} : \text{giovanni}, \text{figlio} : \text{mario}>$$

In ogni caso, anche nei casi in cui non è obbligatorio l'utilizzo dei ruoli, questo può essere utile per aumentare la leggibilità del diagramma.

Relazioni Simmetriche: abbiamo detto che l'utilizzo dei ruoli è obbligatorio se un'associazione insiste più volte su una sola classe e **la relazione non è simmetrica**. Infatti, nel caso in cui la relazione sia simmetrica i ruoli non sono necessari. Ma che cos'è una relazione simmetrica? Si tratta di relazioni per cui:

$$(x, y) \in \text{Relazione} \implies (y, x) \in \text{Relazione}$$

Sono quindi quelle relazioni come l'essere amico, coniuge, collega di lavoro

46.4 Attributi di Associazione:

A volte ci sono degli attributi che non sappiamo dove mettere. Questi attributi sono gli attributi delle associazioni. Ad esempio: ho le classi Studente e Corso, tra loro c'è l'attributo esame. Il voto dell'esame cos'è?

- Non può essere un attributo dello studente, perché uno studente non ha un singolo voto. Ok allora potrei pensare di mettere una lista o un insieme di numeri, che sono comunque valori e non entità, ma poi non saprei a quali esami corrispondono quali votazioni
- Non può essere un attributo di corso, perché non tutti gli studenti che sostengono uno specifico corso prendono lo stesso voto.
- Non può essere un attributo di una nuova classe esame (quindi invece di fare esame come associazione farlo come classe), perché così potrei avere uno studente che fa un esame, e quest'esame potrebbe avere link con più corsi: quindi avrei uno studente che fa un esame che vale per più corsi contemporaneamente. Inoltre questo creerebbe il problema di avere un solo voto possibile per l'esame, per ogni studente che lo sostiene.

Come fare allora? Invece di costruire una nuova classe Esame con attributo Voto, si costruisce la **classe associazione** Esame con attributo Voto: non è una classe, è un'associazione. Posso metterci gli attributi come fosse una classe, le sue istanze però sono coppie, non oggetti, quindi non hanno stati come le entità. Quindi in questo caso l'attributo Voto è la seguente funzione:

$$Voto : Studente \times Corso \in int$$

Quindi è una funzione che associa ad ogni link (nel nostro caso (studente,corso), elemento del dominio $Studente \times Corso$) un valore di un determinato tipo (nel nostro caso un intero, elemento del codominio int). Ciò vuol dire che se una associazione ha un attributo, ogni link istanza di quella associazione avrà un valore per quell'attributo. Quindi, ad esempio:

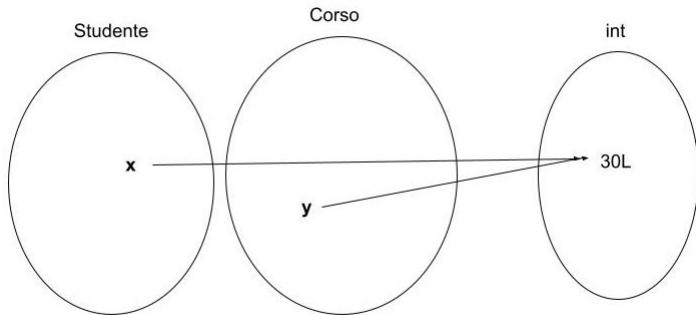


Figure 13: Rappresentazione della funzione Voto calcolata in x e y, cioè $Voto(x,y) = 30L$

$$Voto(x, analisi) = 30 \quad Voto(x, fisica) = 25$$

Due osservazioni:

- La linea spezzata indica sintatticamente la classe associazione
- È vietato legare una classe associazione a una classe

46.5 Molteplicità delle associazioni:

Sappiamo che una associazione non ci dice nulla sul numero di link a livello estensionale. Per sapere qualcosa in merito possiamo definire dei vincoli di molteplicità delle associazioni, ma solo per le associazioni binarie, non n-arie. Le possibili molteplicità sono le seguenti:

- 0..* nessun vincolo (si può omettere)
- 0..1 opzionale, ma al massimo 1. Si tratta di funzioni parziali **VERIFICA**
- 1..* al minimo una
- 1..1 esattamente una. Si tratta di funzioni totali **VERIFICA**
- 0..y ; x..* ; x..Y

Proviamo a fare il seguente esercizio, deducendo noi le molteplicità necessarie: si vogliono modellare gli studenti (con nome, cognome, numero di matricola ed età), il corso di laurea in cui sono iscritti ed i corsi di cui hanno sostenuto l'esame. Di ogni corso di laurea interessa il codice e il nome. Di ogni corso interessa il nome e la disciplina a cui appartiene.

Vediamo che "IL corso di laurea", relativo agli studenti, ci mostra che il vincolo di molteplicità per il numero di corsi di laurea a cui è iscritto uno studente è 1..1. Mentre "I corsi di cui hanno ..." ci fa capire che i corsi sono in numero arbitrario. Dal testo riusciamo a dedurre nessuna limitazione sul numero di studenti minimi o massimi in un corso di laurea o in un corso, quindi 0..*.

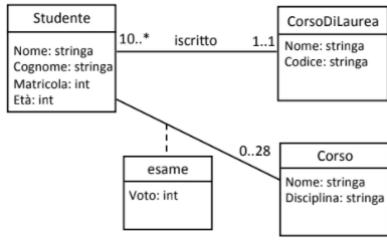
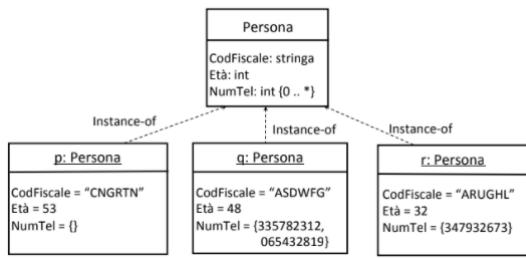


Figure 14: Vediamo che uno studente è iscritto a esattamente un CorsoDiLaurea e può superare al massimo 28 esami. Non ci sono vincoli su quanti studenti possano sostenere un esame, ma c'è il vincolo di minimo 10 studenti per la creazione di un CorsoDiLaurea

46.6 Molteplicità degli attributi:

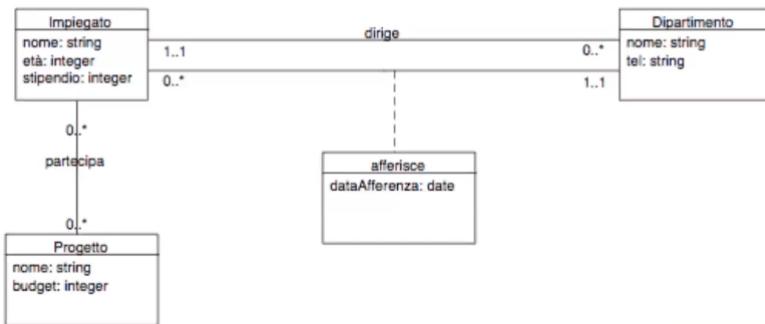
Gli attributi sono funzioni, quindi c'è implicitamente la molteplicità 1..1. A volte però puoi avere degli attributi multivalore: es. di numeri di telefono ne posso avere diversi (uno per casa, uno per ufficio, uno mobile ecc.). Questo significa che se viene specificato un vincolo di molteplicità per gli attributi (che va scritto accanto all'attributo nella forma {x..y}) l'attributo associa ad ogni istanza della classe al minimo x e al massimo y valori per il tipo. Questo viene chiamato attributo multivalore, e formalmente non è più una funzione totale, ma una relazione. Mentre nelle classi l'attributo multivalore è nella forma **tipo \{x..y\}**, nelle istanze si indica con un insieme.



46.7 Esercizio: Dipartimenti Aziendali

Una certa azienda è costituita da diversi dipartimenti, ad ognuno dei quali afferisce un certo insieme di impiegati. Ogni impegno (del quale interessa il nome, l'età, lo stipendio) afferisce esattamente ad un dipartimento. Dei dipartimenti interessa il nome, il numero di telefono, la data di afferenza di ognuno degli impiegati che vi lavorano, ed il direttore. Gli impiegati partecipano a vari progetti aziendali, dei quali interessa il nome ed il budget.

Soluzione: Dipartimenti Aziendali

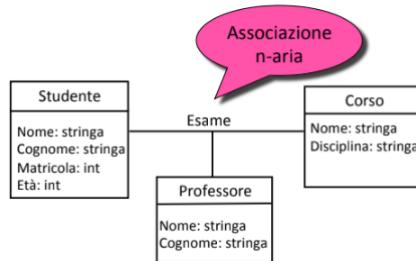


Soluzione: Osservazioni:

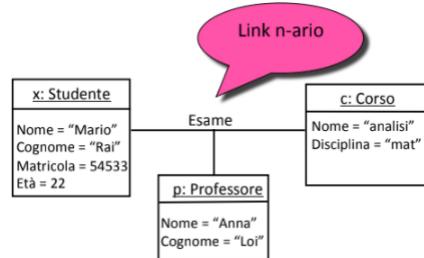
- Notiamo che le date di afferenza non sono un attributo del dipartimento: uno poteva pensare le date di afferenza come un insieme di stringhe, cioè l'insieme delle date di afferenza di ogni impiegato appartenente al dipartimento. Ma se facessimo questi insieme poi non sapremmo a quale impiegato corrisponde quale data, quindi per forza di cose dobbiamo fare una classe associazione: l'attributo data dell'associazione "afferisce a" collega il dipartimento ad ogni singolo impiegato.
- Non modelliamo l'azienda perché essa è il committente. Quello che abbiamo modellato è il sistema informativo dell'azienda. Se avessimo avuto "ci sono aziende ognuna delle quali ha dipartimenti che hanno impiegati ecc." allora avremmo modellato come classe anche Azienda. In questo caso però no: la frase che parla dell'azienda è una frase contestuale, l'azienda non è un'entità ma il committente.

46.8 Associazioni n-arie:

Una associazione può essere definita su n classi, modellando quindi una relazione matematica tra n insiemi:



Le istanze di un'associazione n-aria, cioè i link n-ari, coinvolgono n oggetti: sono quindi n-ple: E



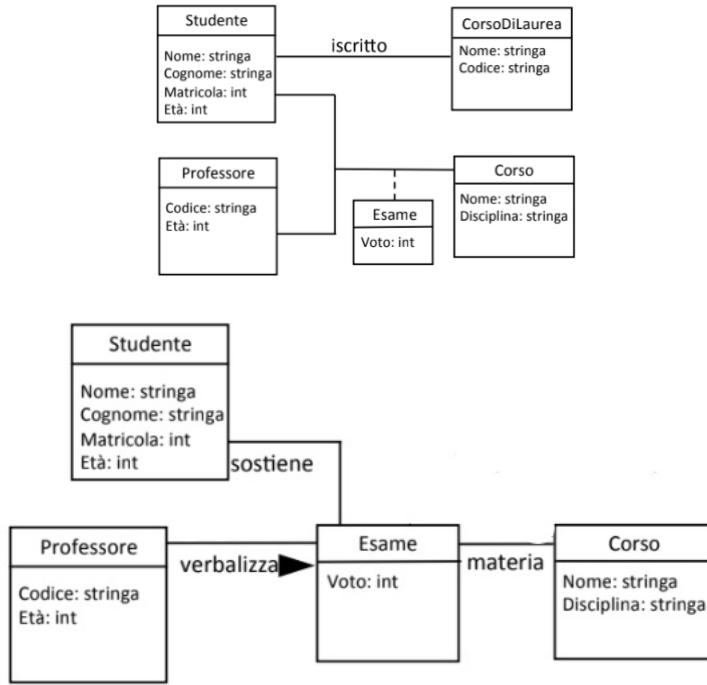
ovviamente, anche le associazioni n-arie possono avere attributi, con le classi associazioni. In UML i vincoli di molteplicità sulle associazioni n-arie ($n > 2$), sono in disuso. Qualora avessimo bisogno di specificare un vincolo di molteplicità lo faremo in linguaggio testuale con un commento, come facciamo nell'esempio seguente.

46.8.1 Esempio:

Si vogliono modellare gli studenti (con nome, cognome, numero di matricola ed età), il corso di laurea in cui sono iscritti, ed i corsi di cui hanno sostenuto l'esame, **con il professore che ha verbalizzato l'esame, ed il voto conseguito**. Di ogni corso di laurea interessa il codice e il nome. Di ogni corso interessa il nome e la disciplina a cui appartiene (ad esempio: matematica, fisica, informatica, ecc.). Di ogni professore interessa codice ed età.

Risposta: Come possiamo intuire si tratta di un esercizio in cui si presenta l'associazione n-aria Esame:

Risposta esatta ma... un'osservazione: Uno potrebbe pensare: "dato che le associazioni n-arie sono più complicate, voglio usare solo associazioni binarie, quindi trasformo l'associazione n-aria in una classe, che avrà associazioni binarie con tutte e tre le classi di prima", così: Ok, ma abbiamo un problema: adesso esame è un'istanza di classe, cioè un oggetto, diciamo $e \in E$, mentre prima era un'istanza di associazione, cioè una tripla, precisamente la tripla (studente x



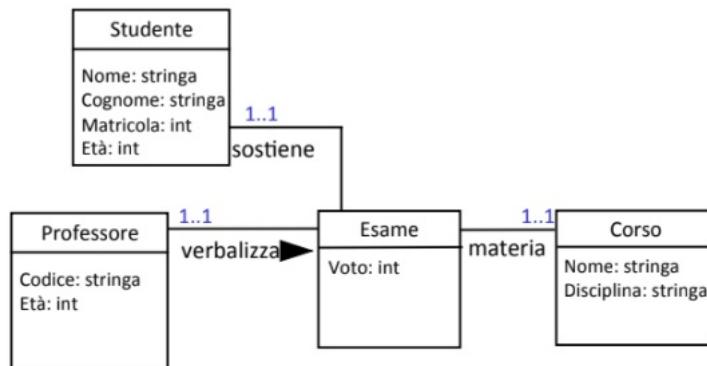
, professore y, corso z). Per far funzionare questo grafico con la classe al posto dell'associazione n-aria, l'oggetto e deve essere in corrispondenza biunivoca con la tripla (studente x , professore y, corso z):

$$e \longleftrightarrow (x, y, z)$$

Iniziamo con \rightarrow , cioè devo essere in grado, dato un oggetto, di saper dire qual è la tupla corrispondente. Per fare ciò mettiamo che:

$$x = \text{sostiene} \quad y = \text{verbalizza} \quad z = \text{materia}$$

Però, ovviamente, ho una sola componente x, una sola componente y e una sola componente z, quindi devo applicare a tutti e tre i vincoli di molteplicità 1..1, perché non posso avere 2 componenti x o 0 componenti z, per esempio: E abbiamo risolto, siamo riusciti a trovare il modo di associare



ad un oggetto la sua tripla corrispondente.

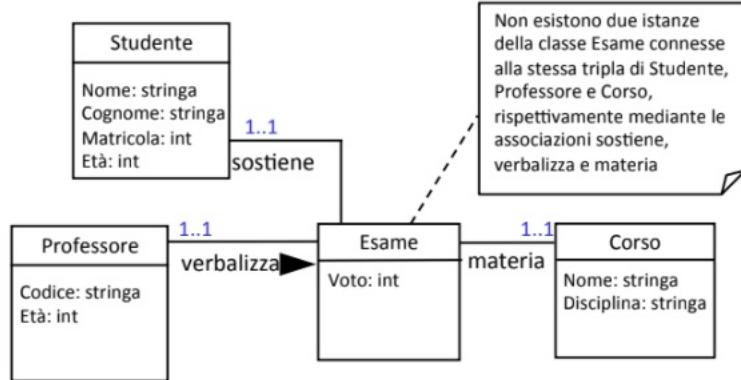
$$e \longrightarrow (x, y, z)$$

Adesso occupiamoci di:

$$e \longleftarrow (x, y, z)$$

Cioè dobbiamo trovare il modo di associare alla tripla l'oggetto, il modo di dire "se mi dai le tre componenti della tripla, io ti so determinare univocamente l'unico oggetto e associato a questa tripla". Ma in realtà, per quello che abbiamo scritto fino ad ora, lasciamo aperta la possibilità di avere un oggetto e' associato ad una tripla con le stesse componenti della tripla associata ad e, questa cosa non la posso accettare. Come eliminare la possibilità di avere due oggetti

che rappresentano la stessa tripla? In UML non è facile, anzi **senza l'utilizzo dei commenti** non è proprio possibile. Un commento è un messaggio in linguaggio naturale che specifica una caratteristica che non si può rappresentare con i meccanismi del diagramma. Nel nostro caso diventa: In questo modo ho la corrispondenza biunivoca tra l'oggetto e e il link (x,y,z), ma a

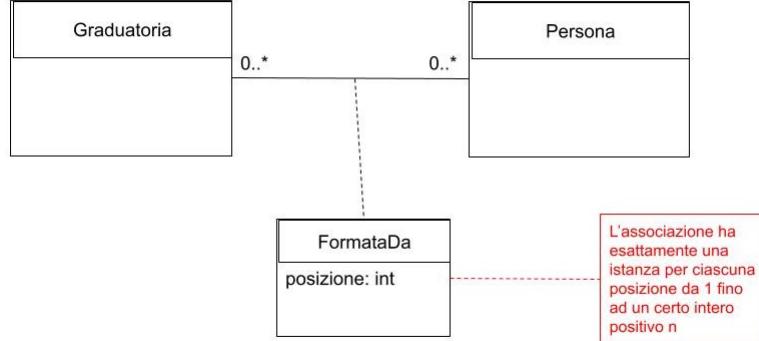


che costo? I commenti in linguaggio naturale sono ambigui e non vanno utilizzati se non sono strettamente necessari, e in questo caso avevamo una soluzione, quella con l'associazione ternaria, che non faceva uso dei commenti; quindi uso l'associazione n-aria. Morale della favola: **usa le associazioni n-arie** se per passare da associazione n-aria a classe instaurando la corrispondenza biunivoca necessiti dei commenti⁵². **Evita in ogni caso l'utilizzo dei commenti.**

⁵²Questa cosa di trasformare associazioni in classi è una cosa che noi umani facciamo continuamente. Il termine tecnico è reificazione: far diventare un'associazione tra due cose una cosa

46.9 Associazioni Ordinate:

Supponiamo di voler considerare una graduatoria di persone. Una graduatoria ha n posizioni, ognuna delle quali occupate da una e una sola persona. Supponiamo inoltre che una persona non possa apparire in graduatoria più di una volta. Una possibile rappresentazione in UML è:



Però, in questo modo necessitiamo di un commento. Quel commento è fondamentale per dire "l'associazione deve avere un'istanza per ogni posizione da 1 a n, senza buchi in mezzo". Questo è ragionevole, dato che, per esempio, non posso avere una graduatoria formata dal 1°, dal 2° e poi dal 10°, ma ho bisogno anche di tutti i classificati da 3 a 9. C'è un altro modo di rappresentare questa situazione, senza usare i commenti: utilizzando l'asserzione **{ordered}**, che va messo sempre dal lato della collezione. In questo modo stiamo dicendo che, data "graduatoria", istanza di **Graduatoria**,

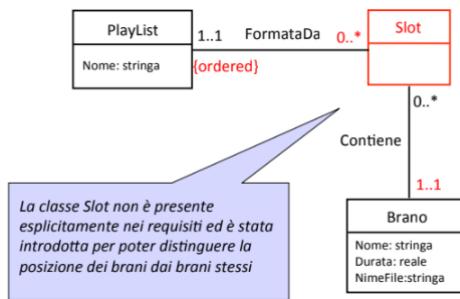


Le istanze dell'associazione **FormataDa** che si legano a "graduatoria", cioè le tuple (persona, graduatoria), sono ordinate. Questo per ogni graduatoria in **Graduatoria**. Questa soluzione è da preferire perché non necessita di commenti (più leggibile) ed è indipendente, astratta rispetto a come verrà mantenuta l'informazione dell'ordine tecnicamente, poiché evita di introdurre uno specifico attributo (nel nostro caso "posizione").

46.9.1 Facciamo un esempio:

Rappresentare in un diagramma delle classi UML playlist costituite da un nome (una stringa) e da un elenco di brani eventualmente ripetuti. Ciascun brano è caratterizzato dal nome (una stringa), la durata (un reale) e il nome del file (una stringa).

Risposta: se non avessimo il requisito "eventualmente ripetuti" questo esercizio ormai sarebbe immediato. Con questo requisito invece dobbiamo ragionare un attimo: come implementare il fatto che un brano possa esserci più volte nella stessa playlist? Un'idea potrebbe essere avere un attributo multivalore di associazione "posizioni", di tipo insieme di interi. Il discorso è che diventa un casino poi mantenere il vincolo per cui ogni posto deve essere occupato e non devono esserci buchi. Ragionando vediamo che stiamo ragionando, anzi predicando, sul POSTO del brano in playlist, non sul brano. Quindi necessitiamo di una classe slot (posto), che è scritta esplicitamente nei requisiti.



Questo esercizio ci fa capire che ci sono delle cose non esplicitamente scritte nei requisiti ma che uno capisce quando analizza da tutti i punti di vista la cosa. In questo modo qui abbiamo capito che non stavamo parlando dei brani ma delle posizioni dei brani, che quindi devono essere una classe del diagramma. Dobbiamo infatti usare gli slot perché dobbiamo ripetere i brani. Se non avessimo dovuto ripetere i brani bastava semplicemente collegare brani e playlist. Il fatto che un brano ripetuto equivalga a un equivalenza superficiale (si tratta di due oggetti che puntano allo stesso brano, non due brani diversi con gli stessi campi dati, perché quest'ultimo sarebbe equivalenza profonda, che non so neanche se avrebbe senso con i brani) impone che non possiamo semplicemente collegare brani e playlist, perché poi avere due brani ripetuti significherebbe avere due link uguali, cioè due coppie uguali per la stessa associazione, e questa cosa sappiamo che non ci può essere. *La soluzione quindi per avere due brani ripetuti nella stessa playlist è quella di usare gli slot come sopra.*

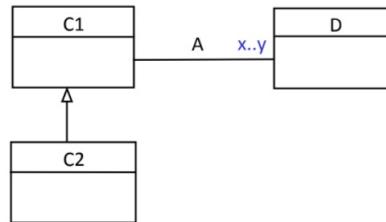
46.10 Generalizzazioni:

Fino ad ora abbiamo sempre avuto classi disgiunte, cioè senza elementi in comune tra loro. In realtà tra due classi può esserci la relazione **is-a**, che matematicamente possiamo pensare come un sottoinsieme. In UML la relazione **is-a** si modella con la **generalizzazione**. La generalizzazione coinvolge una sola superclasse e una o più sottoclassi, e afferma che ogni istanza di ciascuna classe derivata è anche istanza della superclasse (il viceversa in generale non è vero). Quando la sottoclasse è una, la generalizzazione modella la relazione **is-a**. Quindi non è più vero che due classi diverse sono disgiunte (possono essere una figlia dell'altra), ma resta vero che ogni istanza ha **una e una sola classe più specifica**. In UML la relazione **is-a** si indica con una freccia che parte dalla classe figlia e arriva alla classe madre.

46.10.1 Ereditarietà:

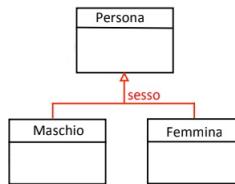
Principio di ereditarietà: ogni proprietà della superclasse è anche una proprietà della sottoclasse (ma non si riporta esplicitamente nel diagramma). In più le sottoclassi hanno anche caratteristiche proprie. In effetti, gli attributi di una classe sono funzioni totali che coprono tutto il dominio, quindi per forza di cose devono coprire anche il sottoinsieme del dominio fatto dalle istanze delle sottoclassi derivate, e quindi anche le classi derivate avranno quegli attributi. Gli attributi specifici di una classe derivata sono anch'essi funzioni totali, ma funzioni totali su un dominio ristretto: quello della funzione derivata. Quindi, ad esempio, Libro ha l'attributo Titolo che è una funzione totale sul dominio dei Libri. Quindi LibroStorico, che è una sottoclasse di Libro ed è quindi un Libro, avrà il Titolo. LibroStorico invece ha un attributo proprio, epoca, che è sì una funzione totale, ma sul dominio dei LibriStorici, quindi non apparirà nella classe Libro perché LibriStorici è dominio sottoinsieme del dominio dei Libri.

Ereditarietà sulle associazioni e le molteplicità: Ogni istanza di C_1 è coinvolta in al minimo x e al massimo y istanze dell'associazione A. Poiché ogni istanza di C_2 è anche istanza di C_1 , segue logicamente che ogni istanza di C_2 è coinvolta in al minimo x e al massimo y istanze della stessa associazione A.



46.10.2 Generalizzazione tra più di due classi:

Per ora abbiamo considerato la generalizzazione per modellare una relazione **is-a** tra due classi, una madre e una figlia; ma la superclasse può però generalizzare anche più di una sottoclasse **rispetto ad un unico criterio**. Questo è come si fa in UML:

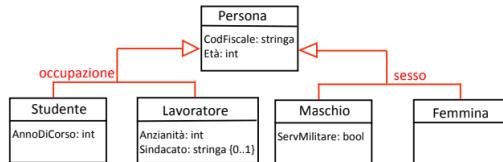


Notare come il criterio rispetto a cui si generalizza - che è il nome della generalizzazione - possa (non obbligatoriamente) essere scritto in rosso accanto alle frecce. Altra cosa da osservare: *questo diagramma afferma che tra Maschio e Persona e tra Femmina e Persona ci sia una relazione is-a*.

46.10.3 Diverse generalizzazioni della stessa classe:

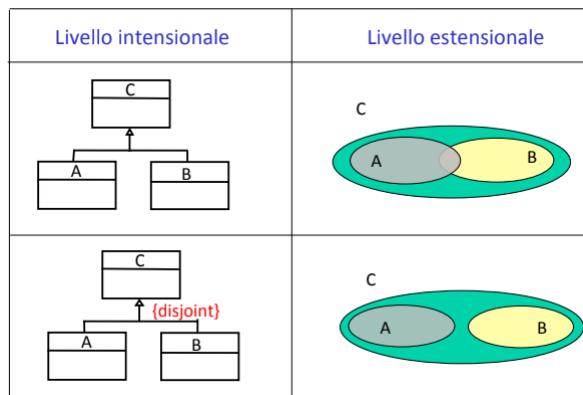
Oltre ad avere una singola generalizzazione con più classi derivate, una superclasse può avere più generalizzazioni. Questo significa che può avere più criteri che la classificano diversamente.

Vediamo un esempio:



Proprio perché rappresentano criteri diversi di classificazione delle istanze della superclasse *non c'è nessuna correlazione tra diverse generalizzazioni*.

Generalizzazioni disgiunte: quando le sottoclassi sono disgiunte a coppie, in tal caso si indica con **{disjoint}**. Quindi, se abbiamo una classe C che ha come sottoclassi A e B, sappiamo valere



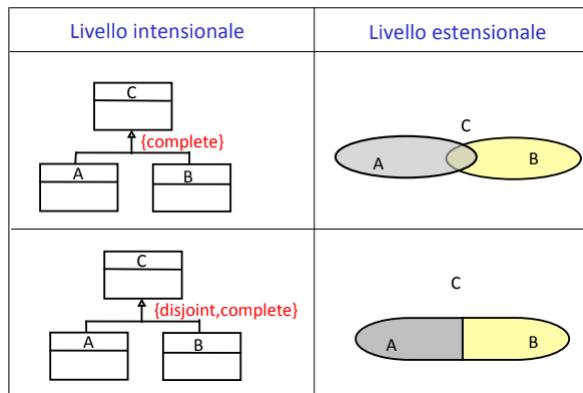
queste relazioni:

$$A \subseteq C \quad B \subseteq C$$

Se la generalizzazione tra queste classi è definita disgiunta con **{disjoint}**, allora vale anche la seguente relazione:

$$A \cap B = \emptyset$$

Generalizzazioni complete: quando l'unione delle istanze delle sottoclassi è uguale all'insieme delle istanze della superclasse. In tal caso si indica con **{complete}**. Quindi, riprendendo l'esempio



di prima relativo a C, A e B, una generalizzazione completa modella la seguente relazione:

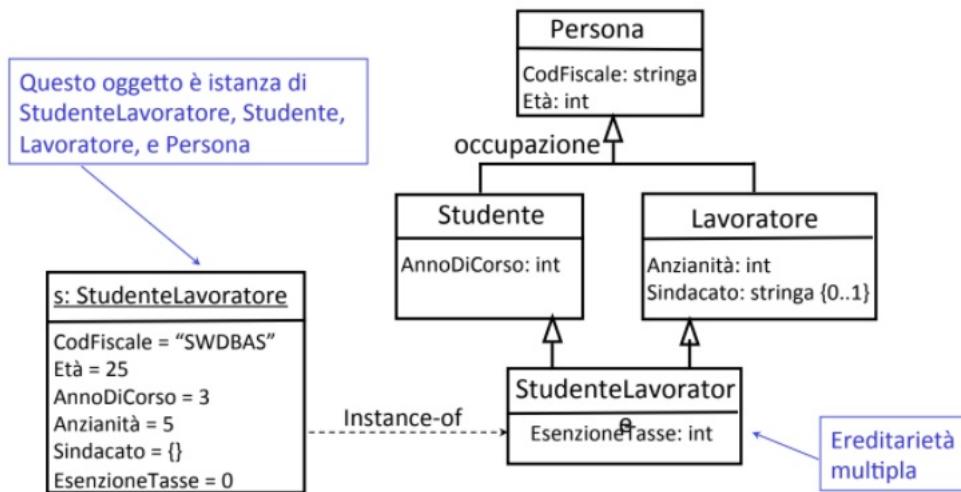
$$A \cup B = C$$

Se, oltre ad essere complete, la generalizzazione è anche disjoint, allora **A e B sono una partizione di C**. Poiché sono disgiunti e la loro unione fa tutto l'insieme C.

Classi Astratte in Java e Generalizzazioni Complete in UML: in Java, la caratteristica principale delle classi astratte è che non ci puoi fare new, non puoi istanziarle direttamente. Puoi solo costruire istanze delle sottoclassi di una classe astratta, ciò significa che le classi astratte sono Complete. Questo perché in UML se hai una generalizzazione completa tu per forza di cose andrai a istanziare una delle sottoclassi e non direttamente la classe madre. Idem nelle classi astratte di Java: non puoi istanziare direttamente una classe astratta ma solo le classi concrete che sono figlie della classe astratta.

46.11 Istanze Comuni ed Ereditarietà Multipla:

Ragioniamo, abbiamo detto che una *istanza ha una e una sola classe specifica*. Allo stesso tempo però abbiamo parlato di classi non disgiunte, sullo stesso livello gerarchico. Ciò significa che quest'ultime in qualche modo dovrebbero avere istanze in comune; Come risolvere questa apparente contraddizione? Semplice: *due istanze NON DISGIUNTE possono avere istanze comuni solo se hanno una sottoclasse comune*. Per avere una sottoclasse comune deve esserci **Ereditarietà Multipla**. Ad esempio:



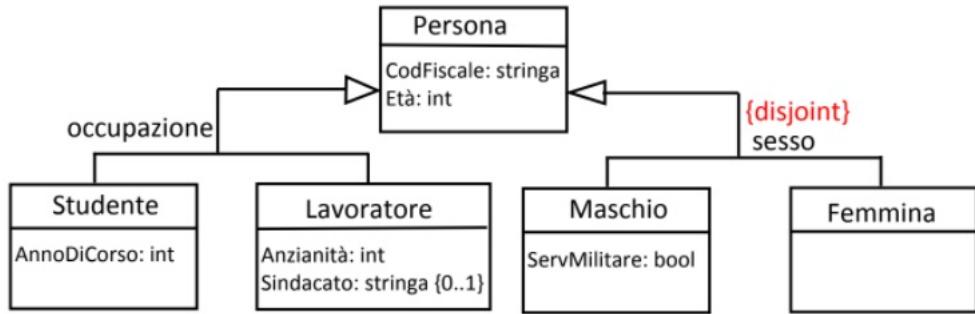
Vediamo che **StudentWorker** è figlia sia di **Student** che di **Worker**. In questo modo **s**, è istanza sia di un'istanza di **Student** che di una di **Worker**.

Evidenziamo il fatto che solo le classi NON DISGIUNTE possono avere istanze comuni. *Classi mutuamente disgiunte non possono avere sottoclassi comuni, e quindi non possono avere istanze comuni*. Questo perché quello che facciamo per fare istanze comuni è creare una sottoclasse di entrambe le classi, cioè **stiamo creando un sottoinsieme dell'intersezione tra i due insiemi che rappresentano le due classi**. Il problema è che le classi disgiunte hanno l'intersezione vuota, quindi non posso creare un sottoinsieme dell'intersezione.

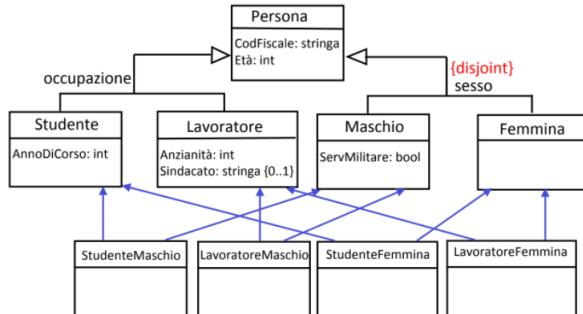
Classi in Java e Generalizzazioni Disjoint in UML: In Java le classi sono tutte disjoint, perché non c'è ereditarietà multipla. Non posso avere infatti istanze comuni tra due classi, se non quelle tra classe madre e classe figlia. Le interfacce in Java invece, avendo l'ereditarietà multipla, possono essere non disjoint. Infatti posso avere una classe che implementa diverse interfacce, e che quindi ha istanze che sono comuni a tutte le interfacce, attraverso la classe di cui è istanza più specifica.

46.11.1 Facciamo un esempio:

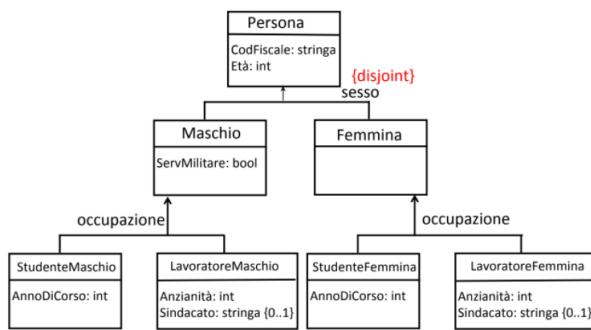
Questo è il grafico: Vediamo che ci sono dei problemi: non posso costruire istanze di **StudentiLavoratori** così! Perché quando vado a fare l'istanza, accanto all'object identifier, dopo ":" devo metterci una classe, quale ci metto? **Studente** o **Lavoratore**? Come detto prima, abbiamo bisogno di una sottoclasse sia di **Studente** che di **Lavoratore**, sottoinsieme dell'intersezione tra questi due. L'istanza avrà quindi **StudenteLavoratore** come classe specifica. Idem per **StudenteMaschio**, **LavoratoreMaschio**, **StudenteFemmina** e **LavoratoreFemmina**. Aggiungiamo allora tutte queste sottoclassi:



In questo modo siamo riusciti a istanziare tutto ciò che volevamo... quasi tutto in realtà: non



abbiamo creato la sottoclasse **StudenteLavoratore**. Ovviamente se serve va creata, ma questo ci permette di parlare di questa cosa: in diagrammi complessi, adottare questo metodo (che è quello giusto da adottare eh) comporta disegnare tantissime sottoclassi, e questo può risultare scomodo; la soluzione allora è disegnare solo le sottoclassi che servono, e le altre non disegnarle, lasciando un commento del tipo "ammetti anche tutte le altre classi che non ho disegnato". Ripetiamo che questo è il metodo corretto di risolvere questo problema, ma a qualcuno potrebbe venire in mente di risolverlo così: Questo secondo modo è **SBAGLIATO**, non farlo così. Il motivo principale per cui



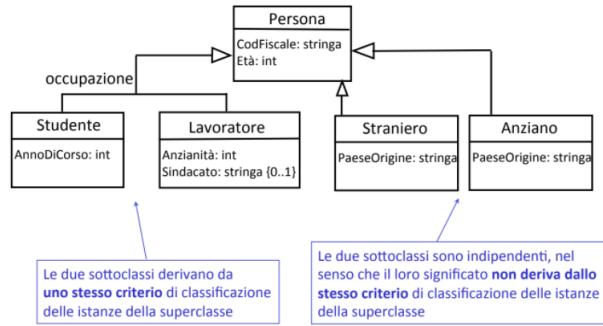
questo metodo è errato è che non sono in grado di specificare che se faccio l'unione dei Lavoratori Maschi e dei Lavoratori Femmine ottengo tutti i Lavoratori, idem per gli Studenti, eppure questa cosa è importante e va inserita. Quindi non posso farla in questa maniera, ma devo adottare il primo modo.

46.12 Differenza tra due is-a e una generalizzazione:

Come abbiamo detto una generalizzazione tra una classe madre e due sottoclassi implica che ognuna delle due sottoclassi sia in una relazione is-a con la classe madre. Questo però non significa che fare una generalizzazione sia la stessa cosa di fare due is-a, e lo dimostriamo con il seguente esempio:

46.13 Specializzazione:

Una sottoclasse non solo può avere proprietà proprie non ereditate dalla superclasse, ma può anche **specializzare** le proprietà che eredita dalla superclasse. Precisamente può:

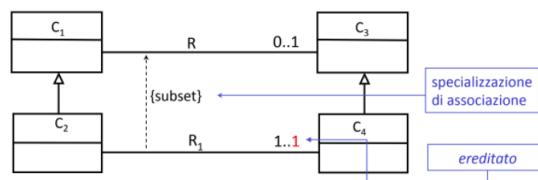


- Specializzare un attributo: data una classe C_1 con un attributo A di tipo T_1 , la classe C_2 , sottoclasse di C_1 , può specializzare A, assegnandogli un tipo $T_2 \subseteq T_1$.



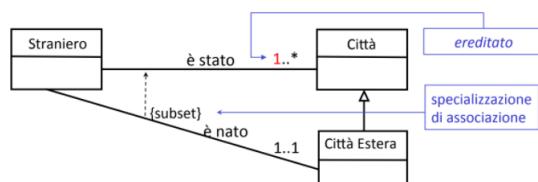
- Specializzare un'associazione: data una classe C_1 , che partecipa ad un'associazione R con un'altra classe C_3 , la classe C_2 , sottoclasse di C_1 , può specializzare R in questa maniera:
 - Definisco una nuova associazione R_1 tra C_2 e una classe C_4 sottoclasse di C_3 (eventualmente C_3 stessa).
 - Stabilisco una dipendenza di tipo **{subset}** tra R e R_1 . **{subset}** ci sta dicendo che "ogni tupla che compare in R_1 deve comparire anche in R".
 - Definisco eventualmente un **vincolo molteplicità massima più specifica** su R_1 rispetto a quelle su R. Cioè su R_1 posso mettere una vincolo di massimo minore o uguale rispetto a quello di R. Non posso allargare il vincolo di molteplicità! Questo perché, se **{subset}** impone che ogni tupla che compare in R_1 deve comparire anche in R, avere una molteplicità massima maggiore in R_1 significherebbe avere più tuple in R_1 che in R, e quindi non tutte sarebbero anche in R, e questa è una contraddizione!

Quindi, graficamente:



46.13.1 Quando si eredita il vincolo di molteplicità minimo nella Specializzazione di Associazioni?

Ok quindi abbiamo visto che si eredita sempre il vincolo di molteplicità massimo, ma invece quello minimo? Facciamo un esempio per vedere: A parole: se l'associazione e l'associazione specializzata

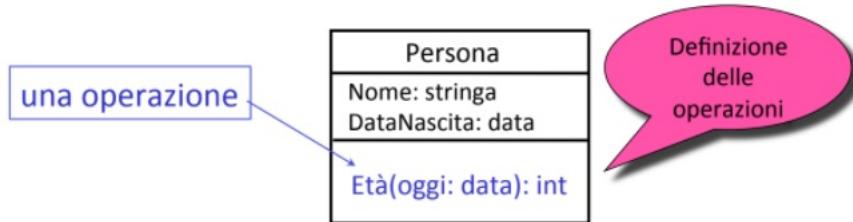


insistono sulla stessa classe la **molteplicità minima viene ereditata al contrario, cioè è la superclasse a ereditare il minimo dalla sottoclasse**. In questo caso infatti la sottoclasse impone che tu sia nato in esattamente una città, è ovvio quindi che questo implichi che tu debba aver visitato almeno una città: quella in cui sei nato. Perciò **la superclasse eredita dalla sottoclasse la molteplicità minima**: il vincolo di molteplicità minima deve essere maggiore o uguale a 1, in questo caso è proprio 1. Non può allargare ma solo restringere il vincolo!

Quindi la specializzazione di associazione prevede che la sottoclasse erediti SEMPRE dalla superclasse la molteplicità massima, mentre prevede che la superclasse erediti NEL CASO DI ASSOCIAZIONE E ASSOCIAZIONE SPECIALIZZATA CHE INSISTONO SULLA STESSA CLASSE dalla sottoclasse la molteplicità minima.

46.14 Operazioni:

Oltre alle proprietà statiche (attributi e associazioni), le classi possono avere anche *proprietà dinamiche*, che in UML sono le **operazioni**. Un'operazione (detta anche metodo) associata a una classe C permette di computare, cioè elaborare sugli oggetti istanze della classe per calcolarne le proprietà o per effettuare cambiamenti di stato (cioè modificare le proprietà). Graficamente un'operazione si mostra così: Quindi vediamo che una classe è definita specificando solo la segnatura



(nome dell'operazione, nome e tipo/classe⁵³ dei parametri e tipo/classe dell'eventuale risultato); quindi non c'è scritto cosa fanno, cioè non c'è scritto il metodo. Una operazione viene invocata su una istanza della classe alla quale appartiene. Questa istanza viene chiamata oggetto di invocazione. Perciò nell'attivazione di un'operazione, oltre ai parametri, c'è sempre implicitamente in gioco l'oggetto di invocazione.

Nota: le operazioni che definiamo sono quelle significative per il modello di analisi, cioè le operazioni che caratterizzano concettualmente la classe; le operazioni orientate alla realizzazione (es. operazioni per gestire gli attributi, come `getAttributo()` o `setAttributo()`), non devono essere definite nella fase di analisi.

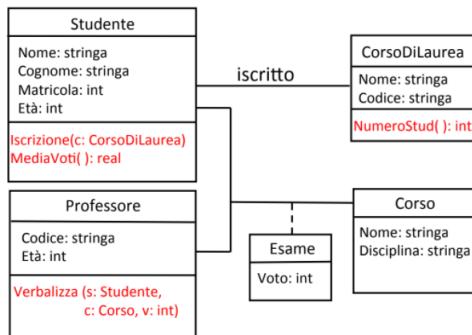
46.14.1 Facciamo un esempio:

Si vogliono modellare gli studenti (con nome, cognome, numero di matricola ed età), il corso di laurea in cui sono iscritti, ed i corsi di cui hanno sostenuto l'esame, **con il professore che ha verbalizzato l'esame, ed il voto conseguito**. Di ogni corso di laurea interessa il codice e il nome. Di ogni corso interessa il nome e la disciplina a cui appartiene (ad esempio: matematica, fisica, informatica, ecc.). Di ogni professore interessa codice ed età. **Al momento dell'iscrizione, lo studente specifica il corso di laurea a cui si iscrive. Dopo l'effettuazione di un esame, il professore comunica l'avvenuta verbalizzazione dell'esame con i dati relativi (studente, corso, voto).** La segreteria vuole periodicamente calcolare la media dei voti di uno studente, e il numero di studenti di un corso di laurea.

Risposta: vediamo che in rosso ci sono nuove cose rispetto all'esercizio già fatto. In particolare vediamo che queste cose nuove parlano di un aspetto dinamico, infatti hanno frasi come "al

⁵³Notare che tutti i tipi o classi (cioè oggetti) coinvolti in una segnatura di operazione devono essere presenti nel diagramma. Questo significa che se il tipo di un parametro, o del ritorno dell'operazione è in realtà un oggetto (quindi più che tipo è una classe) allora la classe di cui è istanza l'oggetto deve essere presente già nel diagramma da qualche parte

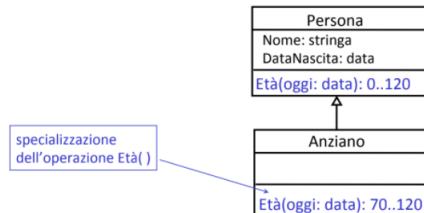
momento dell'iscrizione", "dopo l'effettuazione", "periodicamente". Questo significa che si tratta di operazioni. Rappresentandole nel grafico otteniamo:



Notiamo che le frasi relative alle operazioni non c'entrano nulla con il diagramma delle classi, perché non parlano della descrizione di uno stato delle classi ma di un **cambiamento dello stato delle classi**. In altre parole: nel diagramma delle classi i dati sono fermi, statici, mentre le operazioni, essendo dinamiche, muovono questi dati. Avremo un'altra parte, fuori dal diagramma delle classi, in cui queste operazioni avranno la descrizione di ciò che fanno.

46.14.2 Specializzazione di Operazioni:

Oltre agli attributi e alle associazioni anche le operazioni si possono specializzare nelle sottoclassi. Una operazione si specializza specializzando i parametri e/o il tipo di ritorno. Ad esempio: In gen-



erale il metodo associato a un'operazione specializzata è diverso dal metodo associato all'operazione originaria nella superclasse.

Parallelismo con l'overriding in Java: L'overriding in Java consiste nella sovrascrittura del corpo dell'operazione in una sottoclasse, qui il corpo non ce l'abbiamo ma possiamo solo restringere il risultato nella sottoclasse.

47 Tipi più complessi:

Nel diagramma delle classi fino ad ora abbiamo usato solo tipi di dato semplici (int, String, ecc.). In realtà possiamo usare anche tipi di dato più complessi, ad esempio quelli definibili attraverso costruttori, esempio:

- Record
- Insieme
- Lista
- Ecc.

Facciamo due esempi:

- Pensiamo all'attributo indirizzo come un tipo record con due campi: "strada" di tipo String e "numero civico" di tipo int.
- Pensiamo all'attributo data come tipo record con tre campi: "giorno" di tipo 1..31, "mese", di tipo 1..12 e "anno" di tipo int.

48 Riassunto: tutti gli elementi del Diagramma delle Classi

Ecco una tabella che riepiloga tutti gli elementi del diagramma delle classi visti, con il loro significato e una nota accanto:

Concetto	Significato	Note
Classe	Insieme di Oggetti	Insieme con operazioni
Oggetto	Elemento della Classe	ha vita propria e unico identificatore
Tipo	Insieme di Valori	un valore non ha vita propria
Attributo	Funzione (relazione se multivalore)	portale da classi/associazioni a tipi
Associazione	Relazione	Sottoinsieme del Prod. Cart.
Relazione is-a	Sottoinsieme di Istanze	Implica ereditarietà
General. disg. e comp.	Partizione	Le sottoclassi partizionano la superclasse
Operazione	Computazione	Segnatura definita nelle classi

49 Metodologia per la Costruzione del Diagramma delle Classi:

Un metodo comunemente usato prevede, partendo dalla lettura e analisi del testo dei requisiti, i seguenti passi:

1. Individua le classi ed eventuali oggetti di interesse
2. Individua gli attributi delle classi
3. Individua le associazioni tra classi
4. Individua gli attributi delle associazioni (classi associazioni)
5. Determina le molteplicità di associazioni e attributi
6. Individua le generalizzazioni. Puoi partire dalla classe più generale e scendere o partire da quella più specifica e salire.
7. Determina le specializzazioni
8. Individua le operazioni e associale alle giuste classi
9. **Controllo di Qualità:** se non è soddisfacente, torna al passo 1 e correggi, modifica o estendi.

49.1 Controllo di Qualità:

Le domande da farsi per controllare la qualità del diagramma delle classi sono le seguenti:

I concetti sono stati modellati nella giusta maniera? Infatti abbiamo due bivi davanti, che sono:

1. **Il concetto è una classe o un attributo?** Per scoprirlo dobbiamo:

- Tenere conto della differenza tra classe e tipo:

	Classe	Tipo
Istanze	oggetti	valore
Istanze identificate	da id	da valore
Uguaglianza	basata su id	basata su valore
Realizzazione	da progettare	predef. o basata su strutt. dati predef.

- Basarci sulle seguenti differenze:

- Un concetto sarà una classe se:

- * Le sue istanze hanno vita propria
 - * Le sue istanze sono identificabili indipendentemente da altri oggetti
 - * Se ha proprietà solo sue, indipendenti da altri concetti
 - * Se su di esso si predica nei requisiti
- Un concetto sarà un attributo se:
- * Le sue istanze non hanno vita propria
 - * Le sue istanze non sono identificabili indipendentemente da altri oggetti, perché hanno senso solo per rappresentare proprietà di altri concetti.
 - * Se su di esso non si predica nei requisiti

Nota che le scelte possono cambiare durante l'analisi, all'insorgere di nuovi requisiti nel testo.

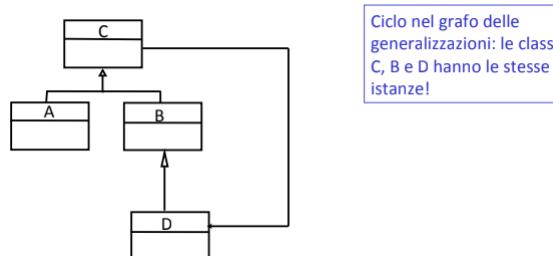
2. Il concetto è una classe o un'associazione? Un concetto verrà modellato come:

- una classe se:
 - Le sue istanze hanno vita propria
 - Le sue istanze sono identificabili indipendentemente da altri oggetti
 - se ha o comunque ha senso pensare ad associazioni con altri concetti
- un'associazione se:
 - Le sue istanze rappresentano n-ple di altre istanze
 - se non ha senso pensare ad associazioni con altri concetti

Nota che anche qui le scelte possono cambiare durante l'analisi, all'insorgere di nuovi requisiti nel testo.

Sono stati colti tutti gli aspetti importanti delle specifiche?

Si è verificato che le generalizzazioni non formano cicli? il grafo delle generalizzazioni non può contenere cicli! Facciamo un esempio per spiegare cosa sono i cicli: Se non ci fosse il

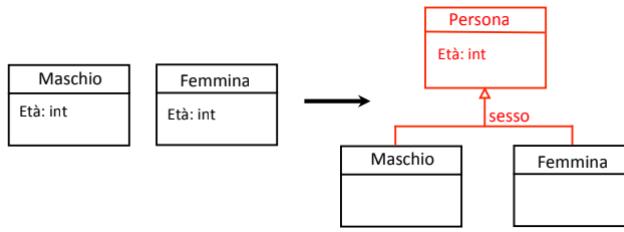


Ciclo nel grafo delle generalizzazioni: le classi C, B e D hanno le stesse istanze!

ciclo infatti potremmo dire che una istanza di B è anche un'istanza di C, essendo B figlia di C; e soprattutto potremmo dire che non è vero il contrario, cioè che non tutte le istanze di C sono istanze anche di B. Il problema è che con questo ciclo invece ci ritroviamo che un'istanza di C è anche istanza di D, che è anche istanza di B, e quindi ogni istanza di B è anche istanza di C: contraddizione!!

Le specializzazioni sono corrette? Per quanto riguarda le specializzazioni di attributo dobbiamo verificare che i valori del tipo dell'attributo della sottoclasse siano un sottoinsieme dei valori del tipo dell'attributo della superclasse. Per quanto riguarda le specializzazioni di associazioni dobbiamo verificare che tra l'associazione R e la sua specializzazione R_1 ci sia una dipendenza {subset} con molteplicità massima uguale o minore su R_1 rispetto a R (e se le due associazioni insistono sulla stessa classe c'è anche il vincolo sulla molteplicità minima).

Si possono applicare ulteriori generalizzazioni? Ad esempio possiamo vedere se due classi condividono degli attributi e quindi possono essere messi sotto una generalizzazione di una superclasse comune. Facciamo un esempio per far capire:



Ci sono classi che sono sottoinsiemi di classi disgiunte?

Part XII

Fase di Progetto:

Ti ricordi la spirale dello sviluppo software? Era divisa in quattro quadranti:

- raccolta requisiti: precedente all'analisi, produce il testo da cui parte l'analisi per vedere cosa dobbiamo fare
- l'analisi, che produce lo schema concettuale: diagramma delle classi, commenti e documenti di specifica (questi ultimi li vedremo quando vedremo le funzionalità). Ci dice in astratto cosa fare.
- progetto e realizzazione: ora ci occupiamo di questo. Si tratta della fase in cui decidiamo come fare ciò che abbiamo deciso di fare nell'analisi.
- verifica e manutenzione, non le vediamo.

Ok quindi progettazione: non andiamo subito a scrivere il codice, questa è l'ultima cosa da fare. Prima si decide che tecnologie utilizzare e come vogliamo realizzare certe cose. Questa è proprio la progettazione, poi viene la realizzazione (scrittura del codice). Quindi, la fase di progetto e realizzazione si occupa di:

- Scegliere la tecnologia usata (per noi sarà sempre Java): tutte le decisioni che si prendono dopo dipendono dalla tecnologia scelta.
- Definire l'architettura del programma: per ogni classe UML avremo una classe Java, per ogni associazione avremo;
- scegliere le strutture dati che utilizzeremo
- produrre la documentazione
- fase di realizzazione: scrivere il codice

Le decisioni prese nella fase di progetto (in inglese design) dipendono dalla prima scelta, cioè dalla tecnologia adottata. Noi in questo corso usiamo Java e le considerazioni che facciamo valgono in larga misura anche per gli altri linguaggi OO.

49.2 Input della fase di progetto:

Si tratta dell'output della fase di analisi. Cioè:

- Schema concettuale:
 - diagramma classi e oggetti (per ora guardiamo solo questo perché vogliamo anticipare la realizzazione, prima vediamo l'analisi con i diagrammi delle classi e poi la loro realizzazione; poi vediamo l'analisi delle attività e poi ritorniamo alla fase di progetto vedendo la realizzazione di questo diagramma; poi torniamo all'analisi per vedere il diagramma degli stati e delle transizioni e poi vediamo la loro realizzazione)
 - diagramma attività (cioè le funzionalità)
 - diagramma stati e transizioni (oggetti reattivi in grado di reagire a eventi esterni)
- Specifica delle operazioni: vedremo più avanti insieme alla realizzazione del diagramma delle attività

49.3 Output della fase di progetto:

Si tratta poi dell'input della fase di realizzazione ed è costituito da:

- scelta delle classi UML che hanno responsabilità sulle associazioni
- scelta delle strutture dati o progettazione di nuove
- scelta corrispondenza tra tipi UML e Java
- scelta gestione proprietà di una classe UML
- progetto dell'API delle classi
- ...

Consideriamo uno per volta tutti i punti. Nota che qua diventa fondamentale la differenza tra astrazione di valore e di entità

49.4 La responsabilità:

Ci chiediamo se, dato un oggetto di una classe, abbiamo bisogno di navigare attraverso un associazione su un'altra classe. Perché sta cosa è importante? Cioè io alla fine in UML posso navigare in tutte le direzioni qualunque classe tramite qualunque associazione. Ok sì vero certo, ma in Java far navigare qualsiasi classe con qualsiasi altra classe è complicato. Dobbiamo verificare se abbiamo l'opportunità di NON navigare in una direzione una determinata associazione perché semplificherebbe di molto il codice. Se un link (istanza associazione) è bidirezionale allora Java non ti dà nessun supporto per farlo: lo devi programmare. Ma se invece tu devi andare solo da una parte, quindi link monodirezionale, allora Java ti dà i riferimenti per farlo. Quindi dobbiamo verificare se ci possiamo appoggiare a quello che ci dà Java o se dobbiamo programmarlo.

49.5 Scelta e Progettazione delle Strutture Dati:

Facciamo un esempio: in un diagramma UML, dato un oggetto persona, hai molte città in cui sei stato: in informatica questo significa che abbiamo una struttura dati per rappresentare una collezione, la collezione giusta è l'insieme di tuple (persona, città). Bene, come rappresentiamo questo insieme? Noi faremo sempre uso del Collection Framework (JCF), perché semplifica il programma ed è efficiente. Java alcune cose le ha fatte in un certo modo: es. un insieme in Java può essere fatto con un array o con una lista. Prendiamo ad esempio l'implementazione con la lista: alcune operazioni saranno semplici altre saranno difficili. Ad esempio cercare un elemento in una lista costerà lineare perché comporterà la scansione della lista. Tu potresti pensare ok lo implemento con la tavola Hash, quindi l'accesso per la ricerca diventa costante ma ogni tanto la tavola hash diventa troppo piccola e quindi bisogna ingrandirla, e quello costa lineare. Noi usiamo la JCF, però in futuro faremo noi le librerie progettando la struttura dati.

49.6 Tipi:

Sono le proprietà locali, in UML stanno dentro gli attributi. Oltre alle classi dobbiamo progettare anche loro. Ovviamente nella maggior parte dei casi avremo già il tipo Java corrispondente a quello UML (es. int), però potrebbe capitare ogni tanto di fare un tipo particolare, es. tipo indirizzo fatto da un record con via, numero, cap e città. Questo te lo devi fare tu, è un'astrazione di valore.

49.7 Gestione delle proprietà:

Ci facciamo domande di questo tipo: questo attributo può essere modificato o no? Questo lo facciamo così non dobbiamo scrivere sia il set che il get, quindi domande per snellire il codice.

49.8 API:

Noi non le scriveremo. Ma in progetti più grandi sì. Sono le segnature delle funzioni che vogliamo realizzare, senza il corpo. L'utente esterno ha bisogno di vedere solo questo.

49.9 Studio di caso: scuola elementare

Tutte ste cose qui ce le fa vedere in pratica, studiando dei casi. Cioè prendiamo dei requisiti, otteniamo lo schema e poi ci iniziamo ad occupare della fase di progettazione. Prendiamo un esempio ora, questi sono i requisiti:

L'applicazione da progettare riguarda le informazioni su provveditorati scolastici, scuole elementari e lavoratori scolastici. Di ogni scuola elementare interessa il nome, l'indirizzo e il provveditorato di appartenenza. Di ogni provveditorato interessa il nome e il codice attribuitogli dal Ministero. Dei lavoratori scolastici interessa la scuola elementare di cui sono dipendenti, il nome, il cognome e l'anno di vincita del concorso.

Esistono solamente tre categorie di lavoratori scolastici, che sono fra loro disgiunte: dirigenti, amministrativi e insegnanti. Dei primi interessa il tipo di laurea che

hanno conseguito, dei secondi il livello (intero compreso fra 1 e 8), mentre dei terzi interessano le classi in cui insegnano, e, per ogni classe, da quale anno.

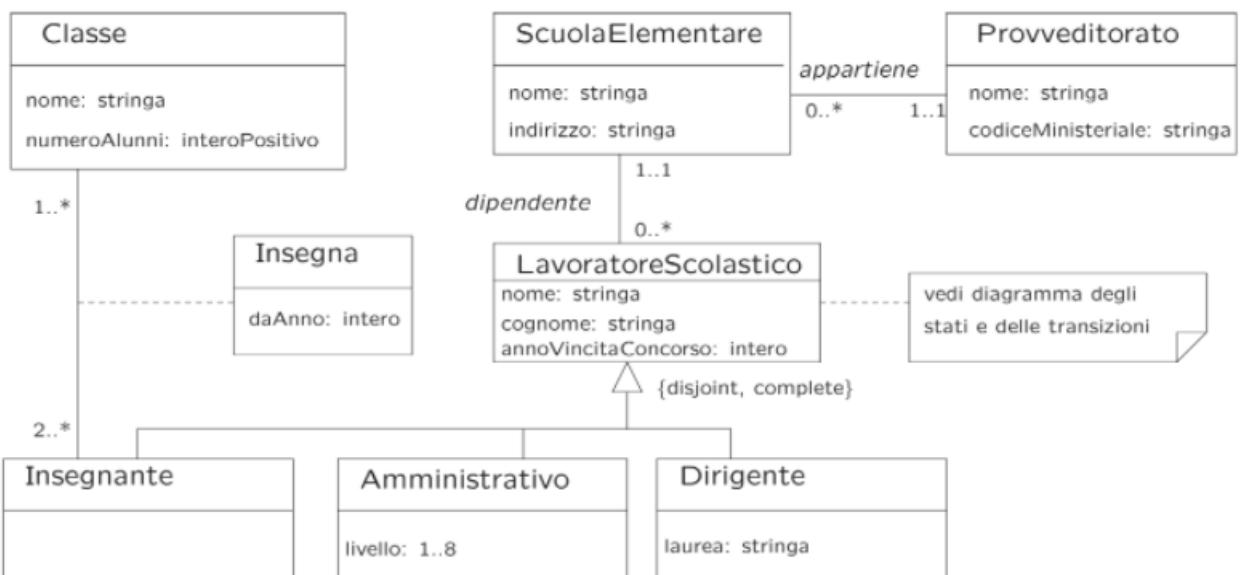
Ogni insegnante insegna in almeno una classe, e ogni classe ha almeno due insegnanti. Di ogni classe interessa il nome (ad es. \IV A") e il numero di alunni.

Abbiamo anche delle informazioni sulle operazioni:

Il Ministero dell'Istruzione deve poter effettuare, come cliente della nostra applicazione, dei controlli sull'insegnamento. A questo scopo, si faccia riferimento alle seguenti operazioni:

- data una classe e l'anno corrente, calcolare quanti insegnanti della classe hanno vinto il concorso da pi' u di 15 anni;
- dato un insegnante, calcolare il numero totale di alunni a cui insegna;
- dato un insieme di insegnanti, calcolare il numero medio di alunni a cui insegnano.

Questi requisiti non faranno parte del diagramma delle classi, perché non parlano di dati ma di manipolazione di dati (a meno che non siano operazioni di classe, ma in questo caso, come nella maggior parte dei casi, non lo sono). Queste sono operazioni esterne alle classi, e faranno parte del diagramma delle attività. Le operazioni non fanno parte del diagramma delle attività ma lo usano. Le operazioni devono essere coerenti con ciò di cui si parla, cioè col diagramma delle classi, che è il nostro alfabeto (nel SW si può parlare solo con ciò che il diagramma delle classi ha definito). Esempio: se io dico, dato una classe dimmi a che scuola appartiene, non c'è un link da classe a scuola. L'analisi mi ha privato di questa possibilità, per farlo deve risentire quelli che hanno fatto l'analisi e farmi sistemare la cosa. L'analisi deve cogliere tutto e solo quello che serve, non si può essere zelanti perché il di più lo paghi. Ok, ora disegniamo il diagramma delle classi: Alcune



osservazioni:

- Notiamo che nei requisiti c'è scritto:

"Esistono solamente tre categorie di lavoratori scolastici"

Quel SOLAMENTE significa che la generalizzazione sarà completa

- Per trovare le giuste molteplicità guardiamo gli articoli: poiché c'è scritto "di ogni scuola elementare interessa [...] il provveditorato" quel IL indica che la molteplicità dei provveditorati nella relazione con i con le scuole sarà 1..1. Idem per la molteplicità delle scuole in relazione ai lavoratori, dato che c'è l'articolo LA. Infine vediamo che i requisiti dicono "ogni classe ha almeno due insegnanti" e "ogni insegnante insegna almeno in una classe", quindi dovremo scegliere le rispettive molteplicità.

49.9.1 Vediamo le responsabilità in questo caso:

Prima di realizzare una classe dobbiamo vedere se ha responsabilità sulle associazioni in cui è coinvolta. Diciamo che **una classe C ha responsabilità sull'associazione A quando, per ogni istanza di C vogliamo poter eseguire operazioni sulle istanze di A a cui l'istanza di C partecipa**. Queste istanze possono avere lo scopo di conoscere le istanze di A, dell'altra classe oppure di modificare attributi o istanze dell'associazione. In altre parole: **una classe ha responsabilità su un associazione se dato un oggetto di quella classe per qualsiasi motivo hai bisogno di attraversare quell'associazione, al fine di avere informazioni sulla tupla istanza dell'associazione**. Come abbiamo detto questo è importante da capire, perché scegliere semplicemente di avere sempre responsabilità dobbia bidirezionale mi vincola a scrivere molto più codice. **Per ogni associazione deve esserci almeno una classe che ha responsabilità su questa associazione**. Vediamo se nel nostro caso ci sono delle direzioni che non vogliamo sapere, e per farlo abbiamo tre criteri:

Criterio delle Molteplicità: Data una scuola elementare devo poter fare controllo di molteplicità e dirti: "ERRORE!!! sono in 2 provveditorati!", oppure "ERRORE!!! non sono in nessun provveditorato!". Per poter fare questo controllo devo poter navigare un'associazione. Quindi questo criterio è facile, basta che ci sia una molteplicità diversa da 0..* e sono costretto a navigare per controllare che il vincolo di molteplicità sia rispettato. **Quindi la responsabilità è logicamente implicata dai vincoli di molteplicità diversi da 0..***.

Criterio delle Operazioni: Dobbiamo sapere le operazioni che vengono svolte su questo diagramma⁵⁴. Noi in questo caso non abbiamo fatto il diagramma delle attività quindi le operazioni ci sono rimaste nei requisiti, perciò li vediamo da lì, ma in generale dovremmo vedere dal diagramma delle attività. Comunque, vediamo che:

- L'operazione relativa a "data una classe e l'anno corrente, calcolare quanti insegnanti della classe hanno vinto il concorso da più di 15 anni" necessita che io sia in grado, partendo da una classe, di navigare l'associazione Insegna, così da poter accedere alle istanze di Insegnante. Quindi classe ha responsabilità su Insegna.
- "dato un insegnante, calcolare il numero totale di alunni a cui insegna" implica che Insegnante ha responsabilità su Insegna
- "dato un insieme di insegnanti, calcolare il numero medio di alunni a cui insegnano" implica che Insegnante ha responsabilità su Insegna.

⁵⁴Il criterio delle operazioni è una cartina tornasole per capire se abbiamo fatto bene il diagramma delle classi: all'esame ci danno solo operazioni che sarà possibile fare, simuliamo mentalmente queste operazioni sul diagramma delle classi per andare a scrivere le responsabilità e intanto capiamo se abbiamo fatto bene il diagramma della classi, perché se scopriamo che un'operazione non si può fare allora abbiamo sbagliato qualcosa nel diagramma delle classi.

Criterio dei requisiti: Essendo i requisiti in linguaggio naturale, e quindi ambigui, è l'ultima spiaggia e conviene, una volta avuti i diagrammi, non andare a riprendere i requisiti. Tuttavia ci sono dei casi in cui è necessario. Esempio: supponiamo che un lavoratore scolastico possa appartenere a tante scuole elementari. Quindi molteplicità $0..*$ a tutti e due i lati. Dato che non c'è nessuna operazione tra le due classi e neanche un vincolo di molteplicità diverso da $0..*$ sembrerebbe che nessuna delle due classi abbia responsabilità sull'associazione, però se fosse così ho scritto un associazione che in realtà non mi interessa, non la voglio navigare. Ma questo è impossibile, un'associazione deve avere sempre almeno un verso di navigazione! Allora ragiono e penso che se ho scritto questa associazione vuol dire che nei requisiti se ne parla, quindi devo tornare ai requisiti e vedere come se ne parla. Sempre in questo esempio che supponiamo, andiamo a vedere i requisiti e c'è scritto: "data scuola interessa provveditorato", ah allora interessa il provveditorato! Ok allora scuola elementare ha responsabilità su associazione con provveditorato.

Ok, adesso possiamo costruire una tabella delle responsabilità.

Associazione	Classe	ha resp.
<u>insegna</u>	' Classe ' Insegnante	SIM SIM R
<u>dipendente</u>	' ScuolaElementare ' LavoratoreScolastico	NO SIM R
<u>appartiene</u>	ScuolaElementare Provveditorato	SIM R NO

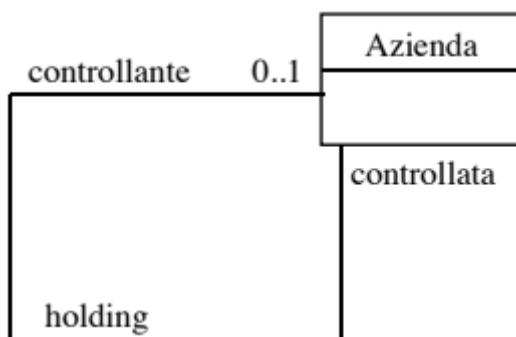
M Markup
O Operazioni
R Requisiti

Figure 15: Il professore chiama R, O e M i requisiti che nelle slide sono rispettivamente 1,2 e 3.

Nota che un'ultima cosa che possiamo fare è un **criterio di verifica**: per vedere se abbiamo fatto bene dobbiamo verificare che per ogni associazione ci sia almeno un "Sì". Cosa tiriamo fuori da questa tabella? Che insegnava ha responsabilità doppia, quindi sarà più difficile da realizzare, dipendente e appartiene più facili.

49.10 Responsabilità sui Ruoli:

Le responsabilità hanno a che fare con la direzione di navigazione, quindi non per forza vanno applicate a un'associazione tra due classi ma possono essere applicate anche ad associazioni che insistono su una sola classe. In questo caso le responsabilità si attribuiscono ai ruoli piuttosto che alle classi: Vediamo in questo caso che, data un'azienda controllata, ci può essere al massimo 1 azienda che la controlla, quindi, per il criterio di molteplicità, l'azienda ha responsabilità sull'associazione holding solo nel ruolo di controllata. Questo perché data un'azienda controllata voglio poter controllare il vincolo di molteplicità, cioè se il numero di aziende che la controllano non supera 1.



50 Strutture Dati:

50.1 Come trovare se e dove servono:

Dobbiamo determinare se avremo bisogno di strutture dati. Dalla tabella delle responsabilità andiamo a vedere di quali classi e rispetto a quali associazioni potrebbero servirci delle strutture dati, e già questa è una prima selezione. Poi passiamo al diagramma delle classi, da cui vediamo che se c'è una molteplicità 0..1 o 1..1 non ho bisogno di progettare strutture dati, ma le molteplicità x..y o x..* me lo impongono. Questo perché devo avere un numero di elementi insieme, quindi per forza di cose devo avere una collezione di elementi. Vediamo nel nostro esempio:

- dato un provveditorato ho bisogno di sapere le scuole elementari? Dallo studio precedente della responsabilità si evince di no, e per fortuna: avendo le scuole appartenenti a un provveditorato molteplicità 0..* bisognava, nel caso in cui ci fosse stata responsabilità, fare una struttura dati - per esempio un insieme - per contenere le scuole associate a un provveditorato. Idem tra scuola elementare e lavoratore
- Diversa è la storia tra classe e insegnante: qui abbiamo che entrambe le classi hanno responsabilità su Insegna. Data un'Insegnante ho bisogno di una struttura dati per contenere la collezione delle sue classi, che sono minimo 1 e massimo *. Data Una Classe ho bisogno di una struttura dati per contenere la collezione dei suoi insegnanti, che sono minimo 2 e massimo *.

Un altro modo per trovare se ci sono delle strutture dati da implementare è quello di vedere le operazioni: nel nostro caso abbiamo ad esempio un'operazione che ci permette, dato un insieme di insegnanti, di calcolare il numero medio di alunni a cui insegnano. Quindi per forza di cose ci serve un insieme di insegnanti in input all'operazione, ce lo dicono i requisiti. Quindi per vedere se servono strutture dati, principalmente guarda queste tre cose:

- Tabella Responsabilità
- Diagramma Classi
- Operazioni, i loro argomenti e i valori restituiti

50.2 Come farle:

Usiamo sempre il JCF. Prima vediamo che interfaccia ci serve (insieme se non dobbiamo avere collezione ordinata, altrimenti lista ecc.) e poi la specifica implementazione.

ESERCIZIO 1 SLIDE N. 22 DELLE DISPENSE IL PROF NON LO HA FATTO

51 I Tipi, da UML a Java:

Dobbiamo vedere tutti i tipi che abbiamo usato in UML (diagrammi, specifica, scelta strutture dati) e poi cercare il corrispondente tipo in Java per rappresentarlo. In genere è opportuno scegliere un tipo base in Java (int, float ...) o una classe Java di libreria (es. String). Ecco una tabella che esplica la corrispondenza da fare tra un tipo in UML e la sua rappresentazione in Java: Ci sono

Tipo UML	Rappresentazione in Java
intero	int
interoPositivo	int
1..8	int
reale	double
stringa	String
Insieme	Set

però due casi in cui i questi tipi base e le classi Java di libreria non bastano:

1. Quando il tipo UML è per un attributo multivalore, es. Attributo NumeroTelefonico: Stringa {0..*}, oppure AnniVincitaConcorsi: intero {0..*}. In tal caso usiamo una collezione. Nota che la JCF non consente la rappresentazione di insiemi di valori di tipi base (int, float...) ma solamente di oggetti, quindi è necessario fare insiemi di Integer, Float...

2. Quando non esiste tipo base o classe di libreria in Java che corrisponda direttamente al tipo, esempio Tipo Indirizzo, che definisco come record di stringhe Via, NumeroCivico e Città. Oppure un altro esempio è Data, voglio farmi un tipo Data (Java ce lo ha di suo, ma io lo voglio rifare). Approfondiamo questo caso insieme ora.

Quindi prendiamo questo secondo caso, nell'esempio di Data. Prima scriviamo il codice poi spieghiamo:

```
// File Tipi/Data.java

public class Data implements Cloneable {

    //COSTRUTTORI

    public Data() {
        giorno = 1;
        mese = 1;
        anno = 2000;
    }

    public Data(int a, int me, int g) {
        giorno = g;
        mese = me;
        anno = a;

        if (!valida()) {
            giorno = 1;
            mese = 1;
            anno = 2000;
        }
    }

    //FUNZIONI GETTER

    public int giorno() {
        return giorno;
    }

    public int mese() {
        return mese;
    }

    public int anno() {
        return anno;
    }

    //CAMPIS FUNZIONE

    public boolean prima(Data d) {
        return ((anno < d.anno)
            || (anno == d.anno && mese < d.mese)
            || (anno == d.anno && mese == d.mese && giorno < d.giorno));
    }

    public void avanzaUnGiorno() {
        // FA SIDE-EFFECT SULL'OGGETTO DI INVOCAZIONE: rende con side-effect l'implementazione
        if (giorno == giorniDelMese())
            if (mese == 12) {
```

```

        giorno = 1;
        mese = 1;
        anno++;
    }
    else {
        giorno = 1;
        mese++;
    }
    else
        giorno++;
}

public String toString() {
    return giorno + "/" + mese + "/" + anno;
}

//CLONE, EQUALS E HASHCODE

public Object clone() { // SERVE LA RIDEFINIZIONE DI clone(), in quanto una funzione
    fa side-effect
    try {
        Data d = (Data)super.clone();
        return d;
    } catch (CloneNotSupportedException e) { //non puo' accadere ma va gestito
        throw new InternalError(e.toString());
    }
}

public boolean equals(Object o) {
    if (o != null && getClass().equals(o.getClass())) {
        Data d = (Data)o;
        return d.giorno == giorno && d.mese == mese && d.anno == anno;
    }
    else return false;
}

public int hashCode() {

    return giorno + mese + anno;
}

// CAMPI DATI
private int giorno, mese, anno;

// FUNZIONI DI SERVIZIO

```

```

private int giorniDelMese() {
    switch (mese) {
        case 2:
            if (bisestile()) return 29;
            else return 28;
        case 4: case 6: case 9: case 11: return 30;
        default: return 31;
    }
}

private boolean bisestile() {
    return ((anno % 4 == 0) && (anno % 100 != 0))
        || (anno % 400 == 0);
}

private boolean valida() {
    return anno > 0 && anno < 3000
        && mese > 0 && mese < 13
        && giorno > 0 && giorno <= giorniDelMese();
}
}

```

Innanzitutto ragioniamo, data è un valore o un'entità? Chiaramente è un'astrazione di valore⁵⁵. Questo è evidente se pensiamo che Data è un tipo. Per esempio potrebbe essere il tipo dell'attributo Nascita, attributo della classe Persona. Se è il codominio di un attributo (che è una funzione che va dalla classe al tipo) è SEMPRE un'astrazione di valore. Solo i tipi possono andare dentro gli attributi, e **i tipi sono i valori**, sempre. Ok quindi data è un valore, dopodiché vediamo che ha una funzione che avanza di un giorno, e lo fa con side-effect. Per l'astrazione di valore con side-effect dobbiamo:

- Ridefinire equals() e hashCode(). Equals() di Object fa la stessa cosa di ==, noi lo dobbiamo ridefinire perché per verificare che due date siano uguali non dobbiamo confrontare gli object identifier ma le sue proprietà (quindi per verificare che due date siano uguali non dobbiamo verificare che i puntatori puntino allo stesso indirizzo ma che Via, NumeroCivico e Città siano uguali). Quando fai astrazione di entità invece non lo devi ridefinire (due oggetti sono uguali solo se sono lo stesso, cioè se hanno lo stesso Object Identifier).
- Ridefinire Clone() siamo davanti a un valore che ha una funzione che fa side-effect. Avendo almeno una funzione con side-effect devo fare clone(). Vediamo che il pattern di clone() è il seguente: va a chiedere il super.clone() quindi chiedo il clone() di Object, che essendo protected non può essere usato dai clienti ma solo chiamato dalle sottoclassi. Quest'ultimo fa un'operazione che può fare solo lui: la copia della zona di memoria in cui è memorizzato l'oggetto. Dopo questo clone() overridato verifica se la classe estende Cloneable, altrimenti interrompe il programma.

⁵⁵I valori sono oggetti matematici, mentre le entità corrispondono a oggetti del mondo reale). Questo può essere utile per intuire se un qualcosa è un valore o un'entità.