

Linguaggi di Programmazione 1

A.A. 2020-2021

Docente: P.A. Bonatti

Dispense tratte dalle Slide del corso di Informatica a cura dello studente **S. Cerrone**

Introduzione

1. Linguaggi di programmazione	10
Breve storia.....	10
Terminologia.....	10
Linguaggi completi.....	10
2. Macchine astratte.....	11
Definizione.....	11
Processore della macchina astratta	11
Tecnologie di realizzazione di una macchina astratta.....	11
Traduttori	12
Supporto a Run Time (SRT).....	13
Compilazione ed esecuzione	13
Proprietà dei linguaggi.....	13
Criteri di scelta del linguaggio	14
3. Paradigmi computazionali.....	14
Definizione.....	14
Tipi di paradigmi.....	14
Conclusioni	14

Paradigma Imperativo

4. Modello imperativo	15
Memoria	15
Assegnazioni.....	15
Modello	15
Ambiente (di esecuzione)	15
Esempi di assegnazione in C	16
5. Data object e legami	17
Data Object.....	17
Legami	17
Il puntatore.....	17
6. Legami di tipo	18
Legame di tipo	18
Type checking	18
Il linguaggio perfetto	18
7. Blocchi di istruzioni.....	18
Necessità e definizioni	18
Ambito di validità (scoping) di legami	19
8. Legami di nome	19
Static scoping.....	19
Mascheramento	20
9. Legami di locazione.....	20
Allocazione statica di memoria	20
Allocazione dinamica di memoria	20
Stack di attivazione.....	21
10. Procedure come astrazioni.....	22
Procedure e astrazione procedurale	22
Dichiarazione e invocazione di una procedura.....	22
Ambiente di esecuzione	22

Esempio	23
11. Record di attivazione	24
Ambiente locale.....	24
12. Propagazione dei data object	24
Realizzazione	24
Ambito statico	25
Ambito dinamico	26
Osservazioni.....	26
13. Parametrizzazione di procedure	27
Parametri.....	27
Associazione dei parametri	27
Parametri IN	27
Parametri OUT.....	27
Parametri IN OUT	28
Aliasing.....	28
Procedure come parametri di procedura.....	28
Macro.....	29
Funzioni	29
14. Implementazione efficiente dell'ambiente non locale con scoping statico	30
Annidamento (nesting).....	30
Rappresentazione variabili non locali.....	30
Visibilità delle procedure.....	30
Mantenimento del vettore degli ambienti non locali	31
Proprietà di questa implementazione	31

Analisi e progettazione O.O.; UML

15. Modello orientato ad oggetti	32
Introduzione	32
Oggetto	32
Metodologie OOAD e UML.....	32
Classe	33
Gerarchie	33
16. Peculiarità della programmazione OO	33
Astrazione con oggetti.....	33
Classi incapsulate.....	33
Scambio di messaggi.....	34
Ciclo di vita di oggetti	34
17. Elementi di UML	34
Diagrammi di classe	34
Associazione	34
Composizioni e Aggregazioni.....	35
Generalizzazioni.....	35
Polimorfismo	35
18. Altri concetti OO.....	36
Classi astratte	36
Visibilità delle componenti di una classe.....	36
19. Diagrammi UML.....	36
UML: richiami	36

Modello statico.....	36
Modello dinamico.....	38
20. Fase di analisi iniziale.....	39
Attività	39
Reperire info.....	40
Enunciato del problema	40
Dominio del problema.....	40
21. Analisi di oggetti e classi	40
Introduzione	40
Astrazioni chiave.....	40
Oggetti e Classi	41
Attributi e metodi.....	41
22. Relazione tra classi	41
Ereditarietà	41
Polimorfismo	42
Classi astratte	42
Associazioni e molteplicità	42
Associazioni complesse	43
Altre forme di associazioni	43
23. Analisi della dinamica del modello	44
Modellazione Dinamica	44
Responsabilità	44
Evoluzione del sistema	44

Java

24. Introduzione a Java.....	45
Tecnologia Java.....	45
Un esempio.....	45
25. JVM e JRE	46
Java Virtual Machine (JVM)	46
Garbage Collector.....	47
Java Runtime Environment (JRE).....	47
Java Runtime Environment e Just in Time Compilation (JRE + JIT).....	47
Sicurezza	47
26. Un po' di sintassi	48
Classi	48
Attributi	48
Metodi	48
Accesso a membri oggetto	49
27. Incapsulazione.....	49
Il problema.....	49
La soluzione	49
28. Costruttori.....	49
Sintassi.....	49
Costruttore di default.....	50
29. File sorgenti, pacchetti e cartelle.....	50
Sorgenti	50
Pacchetti	50

Cartelle e pacchetti.....	51
Fase di rilascio.....	52
30. Ambiente non locale dal paradigma imperativo a quello a oggetti	52
Organizzazione della memoria nei linguaggi di programmazione	52
Ambiente non locale nel paradigma imperativo	52
Ambiente non locale nel paradigma a oggetti	52
31. Identifieri, parole chiavi e tipi primitivi	53
Commenti	53
Blocchi	53
Identifieri	53
Parole chiavi	53
Tipi primitivi.....	53
32. Tipi reference	54
Cosa sono.....	54
Costruzione di oggetti	55
Esempio	55
33. Parametri	56
Passaggio per valore.....	56
Il riferimento <i>this</i>	56
34. Variabili ed espressioni	56
Ambiti	56
Inizializzazioni	56
Espressioni.....	57
35. Casting di primitivi.....	58
Conversioni implicite	58
Conversioni esplicite.....	58
Promozione aritmetica	58
36. Costrutti e controlli.....	59
Enunciati branch.....	59
Enunciati loop.....	59
Controlli speciali	60
Questionario (soluzioni in fondo).....	60
37. Array.....	61
Dichiarazione, creazione ed inizializzazione.....	61
Esempio	62
Multidimensioni	62
Estremi ed assegnazioni	63
Ridimensionamento	63
Copia.....	64
Aiutare il Garbage Collector	64
Questionario (soluzioni in fondo).....	65
38. Ereditarietà	70
Specializzazione	70
Polimorfismo	71
Argomenti polimorfici e operatore <i>instance of</i>	71
Casting	72
39. Altre relazioni tra classi.....	72
Composizioni	72

Aggregazioni	73
Associazioni	73
Molteplicità	73
40. Overloading e Overriding	74
Overloading di metodi	74
Overloading di costruttori	74
Overriding di metodi	74
41. Costruzione di oggetti	76
La parola chiave <i>super</i>	76
Costruzione ed inizializzazione di oggetti	76
42. La classe <i>Object</i> e le classi wrapper.....	77
La classe <i>Object</i>	77
Il metodo <i>equals</i>	77
Il metodo <i>toString</i>	78
Le classi "wrapper"	78
43. Modificatori	78
Generalità	78
Modificatori di accesso	78
Altri modificatori	79
Sommario	81
Singololetto	82
Questionario con spiegazione	83
44. Classi astratte	87
Introduzione	87
Problema A	87
Problema B	88
45. Interfacce	89
Generalità	89
Esempio	89
Vantaggi delle interfacce	91
46. Casting di riferimenti	92
Introduzione	92
Conversioni automatiche	92
Esempi	93
Casting esplicito	93
Questionario (fattill tu)	95
47. String e StringBuffer	96
String	96
StringBuffer	98
48. Garbage collector	98
Funzionamento	98
Esempi di eleggibilità	99
49. Eccezioni (Gestione degli errori)	100
try, catch e (finally)	100
Vincoli	100
Propagazione	100
Definizione	101
Eccezioni catturate	101

Handle or Declare	102
Esempi	102
Nuove eccezioni.....	102
Overriding.....	103
Questionario (soluzioni in fondo).....	103
50. Polimorfismo parametrico vs polimorfismo per inclusione.....	106
Esercizio	106
Una soluzione	106
Una soluzione migliore: template	107
Confronto	108
Note sull'implementazione	109
51. Classi interne	109
Introduzione	109
Gestione degli eventi.....	109
Utilità delle classi interne	110
52. Classe membro di altra classe.....	111
Introduzione	111
Sintassi base	112
Costruzione.....	112
Esempi	113
Modificatori	114
53. Classe definita all'interno di un metodo	115
Introduzione	115
Accesso a variabili locali	115
Esempio	115
54. Classi anonime.....	116
Introduzione	116
Esempio	116

Paradigma Funzionale; ML

55. Introduzione.....	117
Paradigma funzionale	117
ML.....	117
I tipi primitivi in ML.....	117
56. Dichiarazione e scoping in ML	119
Funzioni	119
Altre dichiarazioni di identificatori	119
Scoping	119
57. Tipi strutturati in ML.....	120
Prodotti cartesiani	120
Record.....	120
Dichiarazioni di tipo.....	121
Datatypes e costruttori.....	121
Costruttori con argomenti.....	122
58. Patterns e Matching.....	122
Utilizzo dei costruttori con argomenti.....	122
Funzione che conta gli elementi della lista	122
Definizione per casi	123

59. Liste e Currying	123
Le liste in ML.....	123
Principali operatori sulle liste in ML.....	123
Currying	124
60. Funzioni di ordine superiore.....	124
<i>filter, map, reduce</i>	124
Funzioni anonime	126
Ulteriori dettagli su currying.....	126
61. Polimorfismo parametrico	127
Tipi parametrici.....	127
Type Inference.....	127
62. Encapsulation Interface	128
Signatures	128
Structures	128
Incapsulamento dell'implementazione dei tipi.....	128
Functors.....	128
63. Eccezioni e integrazione con type checking	129
Eccezioni	129
Il costrutto handle	129
Eccezioni con Parametri	130
64. Esempio: un semplice compilatore	130
Elaborazione di simboli in ML.....	130
Definizione del compilatore	131
Generazione del codice	132
Ottimizzazione del codice.....	132
Combinare le fasi con composizione di funzioni	133
Conclusioni	133

Paradigma Logico; Prolog

65. Introduzione.....	134
Paradigma logico	134
Prolog (PROgramming in LOGic).....	134
66. Costrutti Prolog	134
Fatti.....	134
Queries	135
Variabili logiche	135
I termini	135
67. Matching tra query e fatti (sostituzione e unificazione).....	136
Sostituzioni e istanze	136
Unificazione	136
Search tree e Conjunctive queries.....	137
68. Ragionamento	138
Le regole	138
Derivazioni	139
Overloading e Wildcards	140
Regole ricorsive	140
Prolog e l'algebra relazionale	141
69. Liste	142

Sintassi	142
Il predicato member	142
Evanescenza di parametri di Input e Output.....	144
Il predicato append.....	144
70. Applicazioni.....	146
Calcolo simbolico in Prolog.....	146
Calcolo simbolico delle derivate.....	146
71. Programmazione nondeterministica.....	146
In che consiste	146
Applicazione ai giochi	147
Analisi del gioco	150
Riassumendo: caratteristiche uniche di Prolog.....	151

Introduzione

1. Linguaggi di programmazione

Breve storia

Ci sono migliaia di linguaggi di programmazione che, anche se molti sono stati dimenticati, hanno comunque influenzato i linguaggi attuali. Per potersi orientare in questo mare di linguaggi, apprenderli in fretta ed usarli come si deve occorre una comprensione astratta delle caratteristiche dei linguaggi per coglierne somiglianze/differenze e per comprendere lo scopo di ciascun costrutto (ovvero i principi del language design).

Importanti sono i concetti introdotti dai progenitori dei linguaggi d'oggi, dei quali si ricordano:

- **Fortran**: nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output. Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.
- **Cobol**: Inizia a porsi il problema di come rendere i linguaggi più facili da utilizzare, infatti ha una sintassi "English like" (molto "verbosa")
- **Algol60**: indipendenza dalla macchina e definizione mediante grammatica (bakus-naur form), strutture a blocco, supporto generale dell'iterazione e ricorsione.
- **Lisp**: primo vero linguaggio di manipolazione simbolica, paradigma funzionale, non c'è lo statement di assegnazione, e quindi concettualmente non c'è "il valore" ovvero l'idea di cambiare lo stato della memoria. Non c'è differenza concettuale fra funzione e dato: dipende dall'uso. Prima versione essenzialmente non tipata.
- **Prolog**: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile non deterministico (generate and test). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.
- **Simula67**: classe come encapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di dato astratto implementati in Ada e Modula2, e del concetto di classe di Smalltalk e C++.
- **PL/1**: abilità ad eseguire procedure specificate quando si verifica una condizione eccezionale; "multitasking", cioè specificazione di tasks che possono essere eseguiti in concorrenza.
- **Pascal**: programmazione strutturata, tipi di dato definiti da utente, ricchezza di strutture dati. Ma ancora niente encapsulation; si dovrà aspettare Modula.

Terminologia

Il **linguaggio di programmazione** è usato per esprimere (mediante un programma) un processo con il quale un processore può risolvere un problema. Mentre, il **processore** è la macchina che eseguirà il processo descritto dal programma; il processore non va inteso come un singolo oggetto, ma come una *architettura di elaborazione*. Il **programma** è l'espressione codificata di un processo.

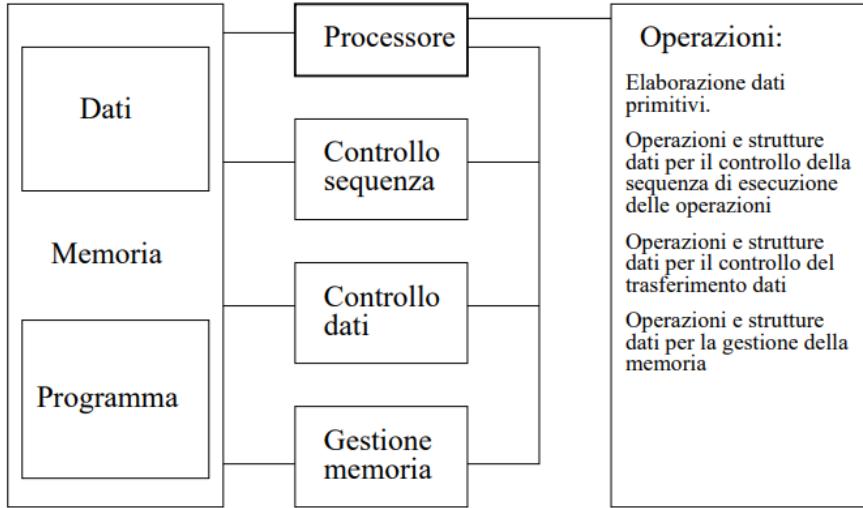
Linguaggi completi

È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli equivalenti in potere espressivo al linguaggio della macchina di Turing e che inoltre riescono ad esprimere anche programmi di cui non è decidibile la terminazione; ad esempio SQL non è un linguaggio completo perché si può sempre dire quando termina il programma, anche se spesso è immerso in linguaggi completi come accade per il linguaggio HTML. Praticamente, se il linguaggio è in grado di simulare arbitrarie macchine di Turing allora lo si definisce completo.

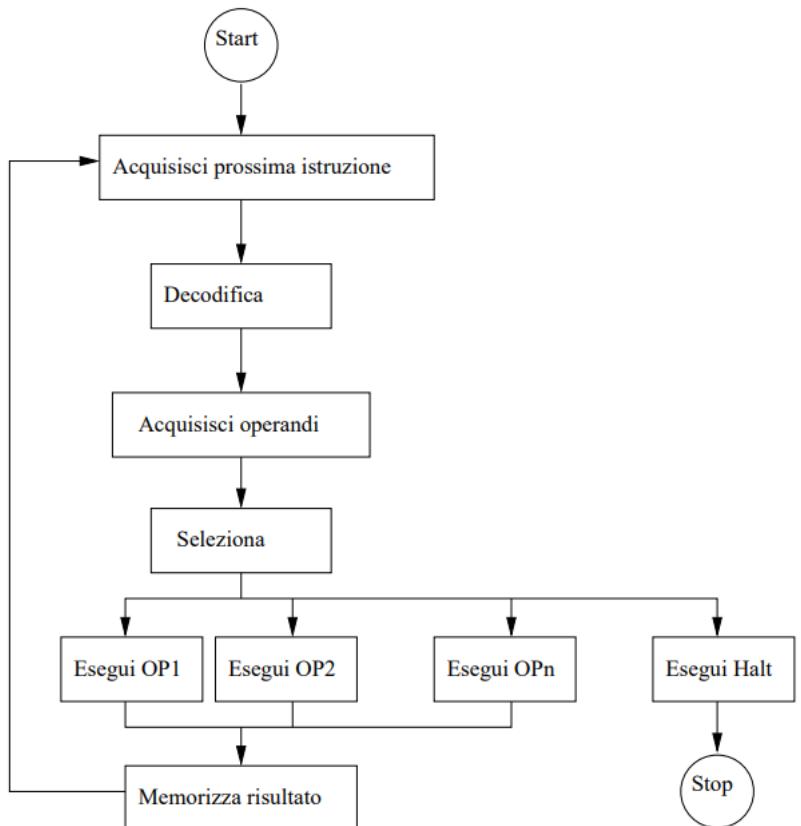
2. Macchine astratte

Definizione

Dato un linguaggio di programmazione L , una macchina astratta per L (in simboli, M_L) è un qualsiasi insieme di strutture dati e algoritmi che permettano di memorizzare ed eseguire programmi scritti in L . La struttura di una macchina astratta è (essenzialmente memoria e processore):



Processore della macchina astratta



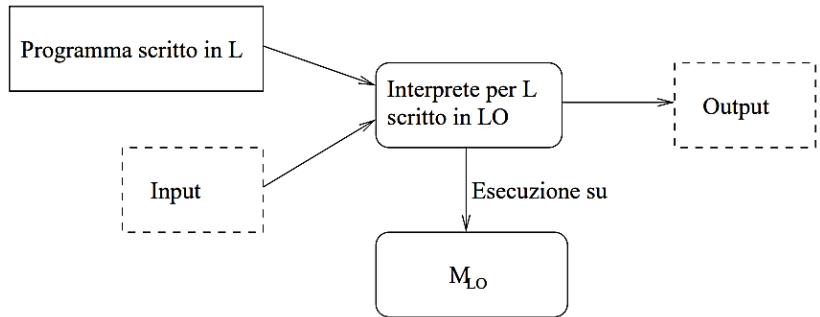
Tecnologie di realizzazione di una macchina astratta

Volendo una macchina astratta la si può realizzare totalmente in **Hardware** (come si era pensato per Lips e Prolog) ma ciò implica la creazione di un Hardware diverso per linguaggi completamente diversi; una soluzione spesso adottata per flessibilità e economicità di progetto è la realizzazione della macchina in **Firmware**, ma oggi è l'utilizzo di **Software** la tecnologia più diffusa per la realizzazione di macchine astratte.

Traduttori

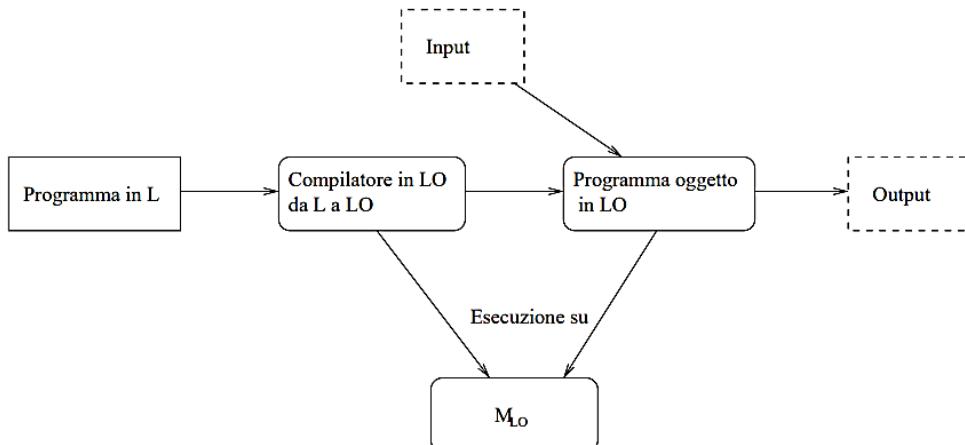
Interpreti: traducono ed eseguono un costrutto alla volta, in favore il debug e la fase di sviluppo hanno un'interazione più snella.

- Interpretazione pura
 - L'interprete è scritto in un linguaggio di solito più basso di quello usato per il programma che è eseguito da una diversa macchina astratta.

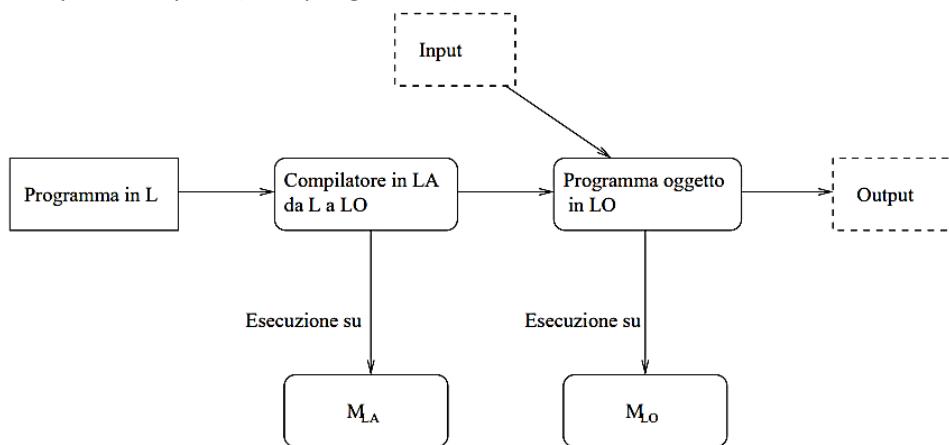


Compilatori: prima traducono l'intero programma; poi il programma oggetto può essere eseguito (anche più volte). Questi controlli fatti in anticipo sono molto importanti poiché, ovviamente, scoprire gli errori a livello di compilazione diminuisce di molto il costo totale del progetto.

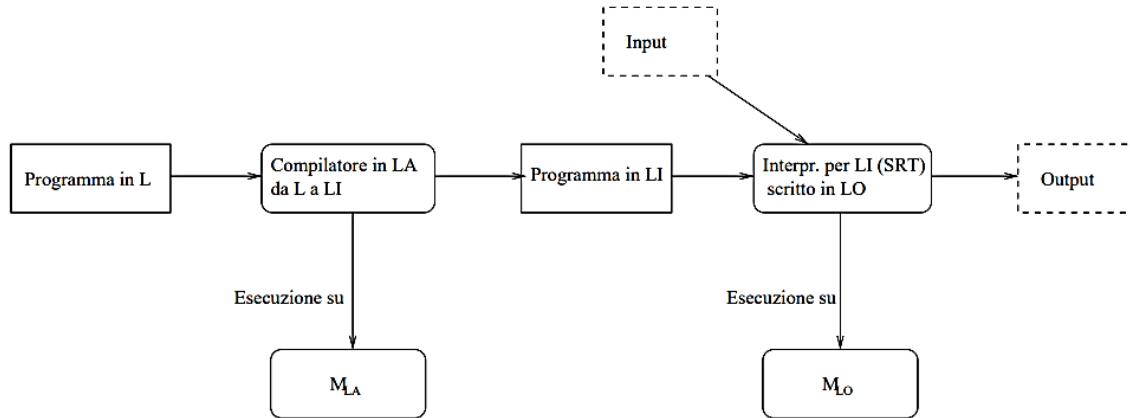
- Compilazione pura (caso semplice)



- Il compilatore è un programma scritto da un diverso linguaggio eseguito da una diversa macchina astratta, qui è l'eseguibile a ricevere l'input e produrre l'output.
- A volte conviene scrivere il compilatore nello stesso linguaggio del programma rendendo così possibile utilizzare un compilatore scritto da un diverso linguaggio per compilare il compilatore completo in *L*.
- Compilazione pura (caso più generale)



- La compilazione passa da un hardware ad un altro, praticamente si usano i tool su una vecchia piattaforma per ricompilerli su una nuova piattaforma. Così da non dover ricostruire i tool e utilizzare i vecchi per una nuova piattaforma.
- Compilazione per macchina intermedia



- Racchiude le caratteristiche dei precedenti.
- Rende possibile implementare i linguaggi senza dover costruire un diverso hardware ma cambiando semplicemente l'interprete, riducendo notevolmente i costi.

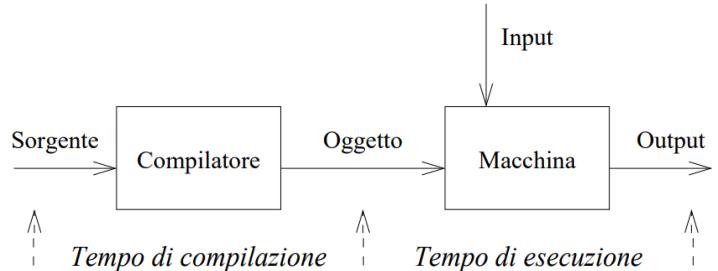
Supporto a Run Time (SRT)

Comporta delle funzionalità aggiuntive (Rispetto a M_{LO}): funzioni di basso livello ed interfacce col sistema operativo (ad esempio funzioni di I/O) e una gestione della memoria. Non necessariamente è una macchina astratta radicalmente diversa, infatti a volte è solo un pacchetto di funzioni o librerie aggiunte automaticamente al codice oggetto.

Compilazione ed esecuzione

Il **tempo di compilazione** è quello che impiega il compilatore per compilare il programma.

Il **tempo di esecuzione** è quello di esecuzione del codice oggetto o, direttamente, il codice intermedio viene interpretato dall'interprete.



Proprietà dei linguaggi

- **Semplicità:** lo scopo è creare un linguaggio quanto più compatto possibile (concisione) senza però renderlo difficile da comprendere (leggibilità)
 - Semantica: minimo numero di concetti e strutture
 - Sintattica: unica rappresentabilità di ogni concetto.
- **Astrazione:** rappresentare solo attributi essenziali per usare il linguaggio
 - Dati: nascondere i dettagli di oggetti
 - Procedure: facilitare la modularità del progetto
- **Espressività:** uguale per tutti i linguaggi di programmazione completi perché tutti sono in grado di programmare tutte le funzioni calcolabili. Naturalmente la facilità di rappresentazione di oggetti (espressività) si oppone alla semplicità.
- **Ortogonalità:** meno eccezioni alle regole del linguaggio.
- **Portabilità:** quanto facilmente il codice può essere portato da una piattaforma ad un'altra senza cambiare comportamento.

Criteri di scelta del linguaggio

Di solito si seleziona il linguaggio da usare in base alla disponibilità di traduttori prediligendo ovviamente il linguaggio conosciuto meglio da parte del programmatore. La comodità dell'ambiente di programmazione e la sintassi aderente al problema sono cause di forza maggiore che non possono essere ignorate a differenza del criterio sulla semantica aderente alla architettura fisica.

3. Paradigmi computazionali

Definizione

un paradigma di programmazione è uno stile fondamentale di programmazione, ovvero un insieme di strumenti concettuali forniti da un linguaggio di programmazione per la stesura del codice sorgente di un programma, definendo dunque il modo in cui il programmatore concepisce e percepisce il programma stesso.

Tipi di paradigmi

- **Imperativo:** Un programma specifica sequenze di modifiche da apportare allo stato della macchina (memoria), praticamente nell'esecuzione cambiano il contenuto della memoria. Se non ci sono assegnazioni o variabili nel programma allora non è imperativo.
- **Funzionale:** Il programma e le sue componenti sono funzioni o meglio espressioni e quindi l'esecuzione diventa una valutazione di funzioni. Nella versione funzionale pura non ci sono variabili né assegnazioni. Quindi non si possono usare cicli e bisogna rimpiazzarli con la ricorsione.
- **Logico:** Programma come descrizione logica di un problema, le sue componenti sono assiomi e l'esecuzione è analoga a processi di dimostrazione di teoremi. I programmi consistono di definizioni di predicati che si comportano come query di un database. Una proprietà del paradigma logico è l'invertibilità: non c'è nessuna distinzione tra input e output, un solo predicato può avere molte funzioni.
- **Orientato ad oggetti:** Programma costituito da oggetti che scambiano messaggi. Non è un vero e proprio paradigma, infatti possono essere istanziati oggetti di tipo imperativo, funzionale o logico.
- **Parallelo:** Programmi che descrivono entità distribuite che sono eseguite contemporaneamente ed in modo asincrono.

Conclusioni

Il paradigma di appartenenza influenza radicalmente il modo in cui si risolve il problema, mentre in linguaggi dello stesso paradigma lo stesso problema ha soluzioni strutturalmente identiche, quindi, imparato a risolvere un problema in un linguaggio, lo si sa risolvere in tutti i linguaggi dello stesso paradigma.

Il paradigma, ovviamente, non è l'unico aspetto determinante in un linguaggio, infatti altri aspetti importanti sono il sistema di tipi supportato, eventuale supporto alle eccezioni, modello di concorrenza e sincronizzazione, etc...

Paradigma Imperativo

4. Modello imperativo

Memoria

Consiste in un insieme di “contenitori di dati” il cui contenuto cambia durante l’esecuzione del programma, ad esempio parole (o celle) della memoria centrale, e tipicamente sono rappresentati dal loro indirizzo associati ai valori in essi contenuti (i valori delle variabili). Dunque (concettualmente) la memoria è una funzione da uno spazio di locazioni ad uno spazio di valori: $mem(loc) = "valore cotenuto in loc"$

Assegnazioni

Tutte le assegnazioni sono definite da un nome che rappresenta la locazione dove viene posto il risultato e un’espressione in cui sono specificati una computazione e i riferimenti ai valori necessari alla computazione.

Durante l’esecuzione il valore dell’espressione va memorizzato nell’indirizzo rappresentato dal nome. Il valore dell’espressione dipenderà dai valori contenuti negli indirizzi degli argomenti dell’espressione rappresentati dai nomi di questi, ottenuto seguendo le prescrizioni del codice associato al suo nome.

Modello imperativo

Il modello imperativo è il più vicino agli elaboratori poiché tutto quello che fa un calcolatore si riduce a operazioni sulla memoria ed il modello imperativo simula le azioni dell’elaboratore a livello di linguaggio macchina. I programmi sono descrizioni di sequenze di modifiche della “memoria” del calcolatore.

Ogni unità di esecuzione consiste in quattro passi (sostanzialmente un assegnamento):

- 1) ottenere indirizzi delle locazioni di operandi e risultato;
- 2) ottenere dati di operandi da locazioni di operandi;
- 3) valutare risultato;
- 4) memorizzare risultato in locazione risultato.

Il modello imperativo si caratterizza per l’uso dei nomi come astrazione di indirizzi di locazioni di memoria.

Ambiente (di esecuzione)

L’ambiente (environment) comprende un insieme di nomi di variabili e parametri (non indirizzi di memoria, piuttosto identificatori) associati a qualcosa da cui si può risalire al valore della variabile o del parametro.

Concettualmente l’ambiente è una funzione da un insieme di identificatori (nomi) a un insieme del codominio che dipende dal paradigma computazionale del linguaggio: $env(id) = ???$

Nel paradigma imperativo, la funzione env associa gli identificatori a locazioni di memoria, le quali, a loro volta, sono associate (funzione mem) al contenuto di memoria: il valore di una variabile x è $mem(env(x))$

Nel paradigma funzionale, non esiste la funzione mem , e la funzione env associa direttamente gli identificatori al valore della memoria.

$env(x)$ è immutabile finché x esiste, nel paradigma imperativo la funzione env identifica una associazione immobile (la locazione di memoria associata a un nome non cambia); anche nel paradigma funzionale puro l’associazione identificatore-valore non cambia.

In una assegnazione $x := x + 1$; la x di sinistra indica la locazione associata al nome (cioè $env(x)$), mentre la x di destra indica il valore della variabile (cioè $mem(env(x))$)

Esempi di assegnazione in C

`x = y;` sx: env(x) dx: mem(env(y))

`int *x, y;
x = &y;` sx: env(x) dx: env(y)

`y = *x;` sx: env(y) dx: mem(mem(env(x)))

`*x` significa "la cella di memoria puntata da x"

cioè "la cella di memoria il cui indirizzo sta scritto dentro x"

cioè "la cella di memoria il cui indirizzo sta nella cella di memoria dov'è allocata x"

`*x = y;` sx: mem(env(x)) dx: mem(env(y))

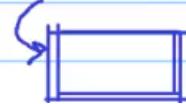
`&y = x;` ERRORE! sarebbe come `env(env(y))` che è malformato!

il C segnala errore di sintassi "not an lvalue"

lvalue corretti: nome di variabile o `*(<pointer expression>)`

`int v[3];`

`env(v)`



`y = v[1];` sx: env(y) dx: mem(env(v) + 1)

`v[1] = y;` sx: env(v)+1 dx: mem(env(y))

`y = x[1] + v[1];` sx: env(x) + 1 dx: mem(mem(env(x))+1) + mem(env(v)+1)

`x[1] = y;` sx: env(x) + 1

come forse `*(x+1)`

`y = *x + *v;` sx: env(v) dx: mem(mem(env(x))) + mem(env(v))

`*v = y;` sx: env(v) (visto come v[0] sarebbe env(v)+0...)

`y = *(v+2);` sx: env(v)+2 dx: mem(env(v)+2)

`y = (*v) + 2` sx: env(v) + 2 dx: mem(env(v)) + 2

`*(x+2) = ...` sx: mem(env(x))+2

ESERCIZI Denota le parti sinistra e destra dei seguenti assegnamenti:

- 1) $x = y[3]$
- 2) Sx: $env(x) + 3$, Dx: $mem(mem(env(y) + 1))$
- 3) $x = y[2]$
- 4) $x = *z + i$ (dove z è un puntatore e i è una variabile di tipo int)
- 5) $*x + 1 = *y$
- 6) $*x + 1 = \&y$
- 7) $*x[1] = y[0]$

$7)(sx: mem(env(x) + 1), dx: mem(env(y) + 1))$ $6)(sx: mem(env(x) + 1), dx: mem(mem(env(y) + 1)))$ $5)(sx: mem(env(x) + 1), dx: mem(mem(env(y) + 1)))$ $4)(sx: env(x), dx: env(y) + 2)$ $3)(sx: env(x), dx: env(y) + 3)$ $2)(sx: env(x), dx: env(y) + 2)$ $1)(sx: env(x), dx: env(y) + 1)$
--

5. Data object e legami

Data Object

Ogni oggetto in un programma viene rappresentato tramite un data object, che praticamente è una quadrupla (L, N, V, T) , ovvero Locazione, Nome, Valore e Tipo. La determinazione di una delle componenti è detta **legame**, quest'ultimo associa uno spazio ad ogni elemento della quadrupla.

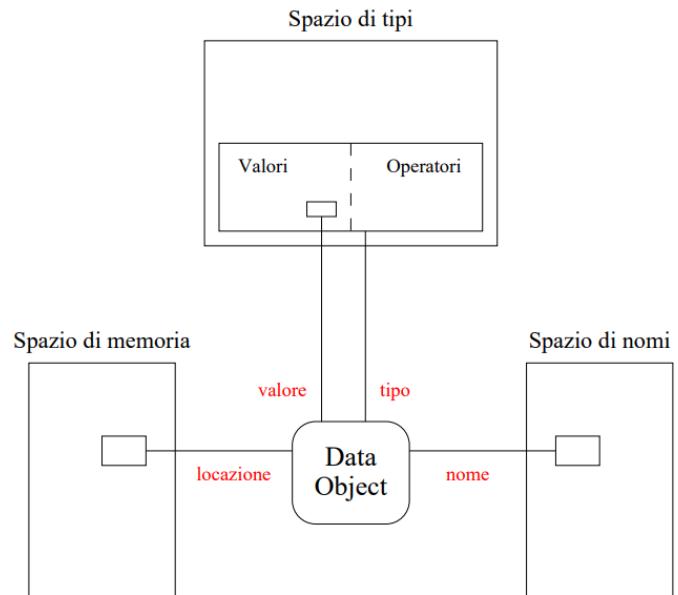
Legami

Ogni legame va in uno spazio diverso (figura)

Legenda usata dal prof: linee in grassetto sono quelle definite a tempo di compilazione, linea singola a tempo di esecuzione. (nel compito se capita un esercizio del genere dire sempre se la linea è di locazione, valore o nome)

Variazioni di legami (*binding*) possono avvenire durante la compilazione (*compile time*), durante il caricamento in memoria (*load time*) oppure durante l'esecuzione (*run time*).

Il **location binding** (legame della variabile alla locazione dove viene posto il nome) avviene durante il caricamento in memoria, oppure a run-time (si veda dopo la gestione dei blocchi);



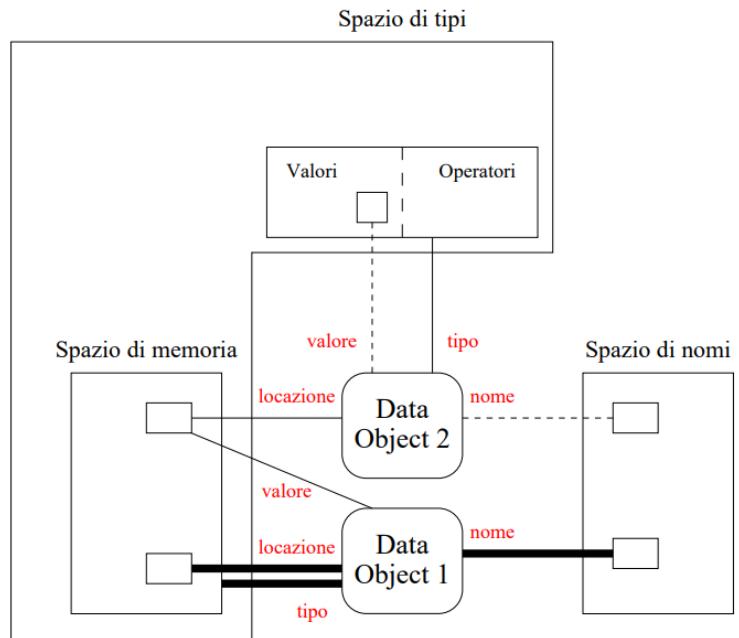
Il **name binding** avviene durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione;

Il **type binding** avviene (di solito) durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione di tipo; un tipo è definito dal sottospazio di valori (e dai relativi operatori) che un data object può assumere.

Il puntatore

Un puntatore è un valore che si riferisce ad un oggetto che si trova altrove, quindi quando il puntatore entra in gioco ci si ritrova con due oggetti. A tempo di compilazione il puntatore riceve un nome, il tipo e la locazione.

Comprende lo spazio di memoria anche come tipo di dato, ovvero il valore della locazione di memoria del secondo oggetto, quest'ultimo avrà comunque un tipo (oltre alla locazione) e potrà o meno avere un legame di nome o di valore inizializzato a tempo di esecuzione (dipende da come è scritto il codice).



La deallocazione è necessaria, perché la modifica del legame di valore genera di solito dati non più accessibili per nome o riferimento. Alcuni linguaggi possiedono meccanismi di recupero automatico di memoria (garbage collector).

6. Legami di tipo

Legame di tipo

Il legame di tipo è per definizione correlato al legame di valore, infatti, sia quando si instaura che quando viene modificato, occorrerebbe controllare (*type checking*) la consistenza con il legame di valore.

Un linguaggio è **dinamicamente tipizzato** se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza (se avviene) avvengono durante l'esecuzione.

Un linguaggio è **staticamente tipizzato** se il legame avviene durante la compilazione; in questo caso il controllo di consistenza (se avviene) può avvenire in entrambe le fasi.

Type checking

Il meccanismo di controllo di consistenza della coppia dei legami valore-tipo è il **type checking**, esso può avvenire durante la compilazione, durante l'esecuzione o non avvenire affatto.

Un linguaggio è **fortemente tipizzato** se il controllo di consistenza avviene sempre: il più possibile durante la compilazione e, negli altri casi, durante l'esecuzione. Un linguaggio fortemente tipizzato è anche staticamente tipizzato (da esempio è il linguaggio Java come vedremo in seguito).

Un linguaggio è **debolmente tipizzato** se il controllo di consistenza può non avvenire affatto in numerosi casi. Il linguaggio C, ad esempio, è debolmente tipizzato, infatti in esso esistono le operazioni di *casting*, che consentono di forzare, in esecuzione, l'interpretazione di un qualunque valore secondo un qualunque tipo (anche un tipo diverso da quello a cui il valore è stato precedentemente associato); esistono *puntatori a void*, che godono, in esecuzione, di conversione di tipo implicita verso qualunque altro tipo puntatore; ed esistono le *union*, che consentono di interpretare una collezione di dati correlati secondo diverse attribuzioni di tipo indipendenti.

Il linguaggio perfetto

Sarebbe considerato un linguaggio perfetto un linguaggio Turing completo, in cui il controllo di consistenza di tipo avvenisse completamente durante la compilazione e in cui il compilatore non generasse più errori del necessario, se esistesse sarebbe staticamente e fortemente tipizzato (il più "forte" di tutti i fortemente tipizzati).

Il linguaggio perfetto non può esistere poiché se esistesse dovrebbe essere capace di decidere la correttezza di un programma generico senza la sua esecuzione, quindi per assurdo dovrebbe essere capace di decidere la terminazione di un programma prima che si arrivi all'istruzione mal tipata (esempio un intero a cui assegno una stringa), questo però renderebbe il supposto linguaggio perfetto non completo secondo Turing, contrastando l'ipotesi.

7. Blocchi di istruzioni

Necessità e definizioni

Si è avuta la necessità di raggruppare in blocchi le istruzioni per ridurre i possibili errori e per meglio definire l'ambito delle strutture di controllo; l'ambito di una procedura; l'unità di compilazione separata; e l'ambito dei legami di nome.

La parte comune dei blocchi che definiscono l'ambito di validità di un nome contengono sempre sezione di dichiarazione del nome ed una sezione che comprende gli enunciati sui quali ha validità il legame.

Al fine dimostrativo definiamo il seguente pseudo linguaggio per rappresentare un blocco:

```
...
BLOCK A;
DECLARE I;
BEGIN A
...
END A;
...
```

Ambito di validità (scoping) di legami

Essenzialmente ci sono due tipi di ambito di validità ovvero lo scoping statico e lo scoping dinamico:

- In **ambito statico** o lessicale.

Blocchi annidati vedono e usano i legami dei blocchi più esterni (legami non locali) e, di solito, possono aggiungere legami locali o sovrapporli a quelli del blocco esterno).

- In **ambito dinamico**

Concetto qui esaminato solo in relazione ai blocchi annidati, ma che assume il proprio senso maggiore quando vi sono procedure chiamanti e chiamate. In questo caso la procedura chiamata vede e usa i legami visti e usati dalla procedura chiamante.

8. Legami di nome

Static scoping

```
PROGRAM P;
DECLARE X;
BEGIN P
...
BLOCK A;
DECLARE Y;
BEGIN A
...
BLOCK B;
DECLARE Z;
BEGIN B
...
END B;
...
END A;
...
BLOCK C;
DECLARE Z;
START C
...
END C;
...
END P;
```



{X da P} {X da P, Y da A} {X da P, Y da A, Z da B} {X da P} {X da P} {X da P, Z da C}

X è conosciuto a livello di A, ma Y non è conosciuto a livello di P, se termina un blocco muore anche la variabile definita in essa.

Mascheramento

Se dentro un blocco interno riutilizzo una variabile definita precedentemente da un blocco esterno allora prevale dentro il blocco innestato la dichiarazione più vicina, ad esempio:

```
PROGRAM P;
  DECLARE X,Y;
BEGIN P
  ...
  {X e Y da P}
  BLOCK A;
    DECLARE X,Z;
  BEGIN A
    ...
    {X e Z da A, Y da P}
  END A;
  ...
  {X e Y da P}
END P;
```

9. Legami di locazione

Allocazione statica di memoria

Si dice allocazione statica di memoria (*load-time*) quando le variabili conservano il proprio valore ogni volta che si rientra in un blocco (il legame di locazione è fissato e costante al tempo di caricamento).

Se il linguaggio prevede ciò, allora, se uso una variabile all'interno del blocco, essa mantiene il suo valore anche se rientro successivamente nel blocco, ad esempio:

```
PROGRAM P;
  DECLARE I;
BEGIN P
  FOR I:=1 TO 10 DO
    BLOCK A;
      DECLARE J;
    BEGIN A
      IF I=1 THEN
        J:=1;           {I da P, J da A}
      ELSE
        J:=J*I;
      END IF
    END A;
  END P;
```

Allocazione dinamica di memoria

Si dice allocazione dinamica di memoria (*run-time*) quando il legame di locazione (e anche di nome) è creato all'inizio dell'esecuzione di un blocco (ogni volta che si entra in un blocco le variabili di quel blocco potrebbero trovarsi in una locazione diversa) e viene rilasciato a fine esecuzione:

L'allocazione dinamica si realizza attraverso il **record di attivazione** di un blocco:

- Un record di attivazione contiene tutte le informazioni sull'esecuzione del blocco necessarie per riprendere l'esecuzione dopo che essa è stata sospesa.
- Può contenere informazioni complesse, ma per realizzare un legame dinamico di locazione in blocchi annidati è sufficiente che contenga le locazioni dei dati locali più un puntatore al record di attivazione del blocco immediatamente più esterno

Stack di attivazione

In ogni momento dell'esecuzione lo *stack di attivazione* contiene i record "attivi":

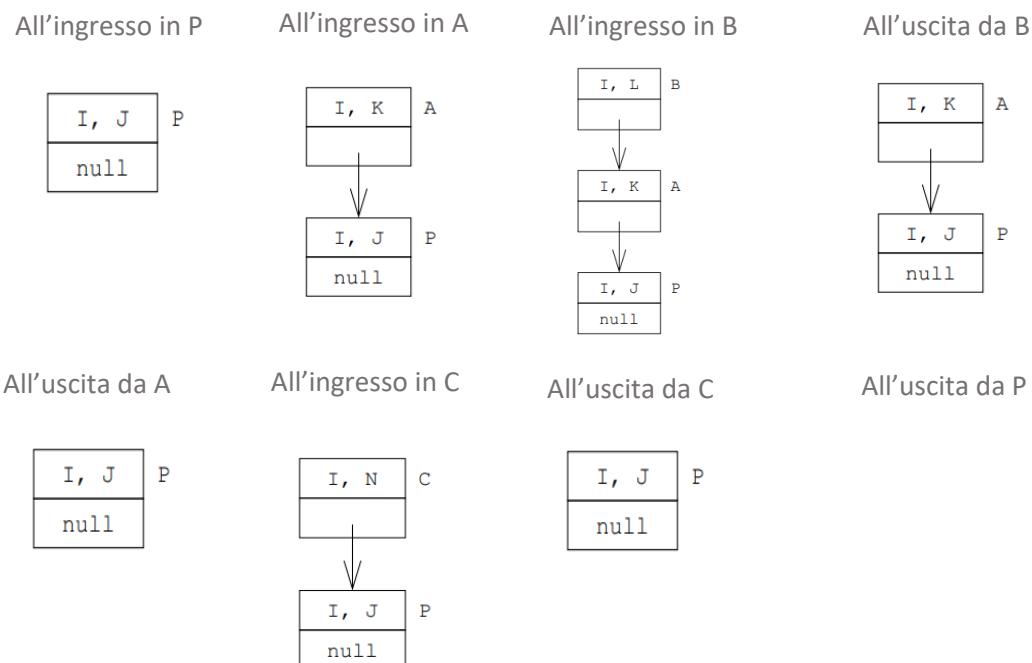
- il top dello stack contiene sempre il record del blocco correntemente in esecuzione;
- ogni volta che si entra in un blocco, il record di attivazione del blocco viene posto sullo stack (**push**);
- ogni volta che si esce da un blocco, viene eliminato il record al top dello stack (**pop**).

```

PROGRAM P;
    DECLARE I,J;
BEGIN P
    BLOCK A;
        DECLARE I,K;
BEGIN A
    BLOCK B;
        DECLARE I,L;
BEGIN B
    ...
    {I e L da B, K da A, J da P}
END B;
    ...
    {I e K da A, J da P}
END A;
BLOCK C;
    DECLARE I,N;
BEGIN C
    ...
    {I e N da C, J da P}
END C;
...
{I e J da P}
END P;

```

Contenuto dello stack di attivazione (gli ingressi sono i **BEGIN**, e le uscite gli **END**):



10. Procedure come astrazioni

Procedure e astrazione procedurale

Le **procedure** sono astrazioni di parti di programma in unità di esecuzione più piccole, come enunciati o espressioni, in modo da nascondere i dettagli irrilevanti ai fini del loro uso e riuso.

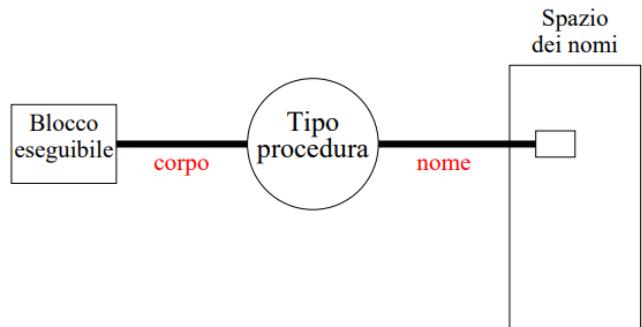
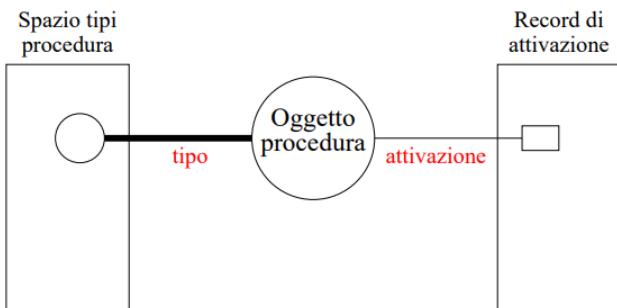
L'utilizzo delle procedure ha i seguenti vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; inoltre la suddivisione dei compiti in ogni brano di programma rende possibile la progettazione top-down.
- Unità di programmi indipendenti o con dipendenze ben specificate a livello più alto.
- Riusabilità di brani di programmi, che implica una riduzione di errori.

Se si definiscono come unità di esecuzione le espressioni, gli enunciati, i blocchi ed i programmi (in ordine crescente di complessità) allora si definisce **astrazione procedurale** la rappresentazione di una unità di esecuzione attraverso un'altra unità più semplice. Praticamente è la rappresentazione di un blocco attraverso un enunciato o una espressione.

Dichiarazione e invocazione di una procedura

La **dichiarazione** di una procedura causa la generazione di un oggetto, il processo avviene durante la compilazione (vedi figura a destra).



(figura a sinistra) Il processo avviene durante l'esecuzione, nel momento in cui c'è l'**invocazione** della procedura. Ogni invocazione diversa della stessa procedura causa la generazione di un nuovo "oggetto procedura" con lo stesso legame di tipo, ma con diverso record di attivazione. Quando una procedura viene chiamata si crea a run-time il record di attivazione.

Ambiente di esecuzione

Analogamente a quanto avveniva per un blocco (di cui la procedura è astrazione), il record di attivazione rappresenta l'intero ambiente di esecuzione di una procedura. Esso consiste di solito in:

- 1) ambiente locale (tutti i data object che sono definiti all'interno della procedura);
- 2) Ambiente non locale (tutti i data object la cui definizione è propagata da altre procedure);
- 3) Ambiente dei parametri della procedura (contiene informazioni sui dati che sono passati dalla procedura o che sono passati alla procedura).

Ogni volta che una procedura viene invocata il suo record di attivazione viene aggiunto al cosiddetto **stack di esecuzione**. Sul top dello stack c'è sempre il record relativo alla procedura correntemente in esecuzione. Alla terminazione della procedura, il record di attivazione viene rimosso dallo stack.

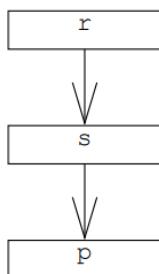
Esempio

Stack esecuzione

p: prima di chiamare s

Stack esecuzione

r: prima di chiamare r



```

program p;
var i;
procedure q;
begin
  ...
end;

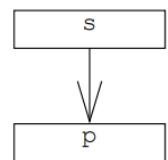
procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
  end;

procedure s;
begin
  r;
  q
end;

begin
  i:= 2;
  s
end.
  
```

Stack esecuzione

s: prima di chiamare r



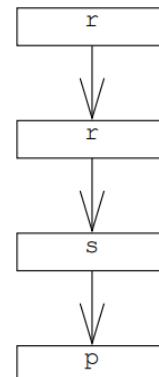
```

program p;
var i;
procedure q;
begin
  ...
end;

procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
  end;
  
```

Stack esecuzione

r: prima di chiamare q



```

procedure s;
begin
  r;
  q
end;
  
```

```

begin
  i:= 2;
  s
end.
  
```

```

program p;
var i;
procedure q;
begin
  ...
end;

procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
  end;

procedure s;
begin
  r;
  q
end;

begin
  i:= 2;
  s
end.
  
```

```

program p;
var i;
procedure q;
begin
  ...
end;

procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
  end;
  
```

```

procedure s;
begin
  r;
  q
end;

begin
  i:= 2;
  s
end.
  
```

q: prima di terminare

r: prima di terminare

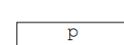
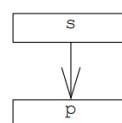
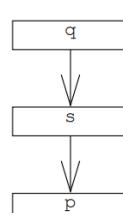
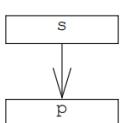
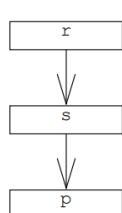
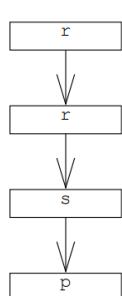
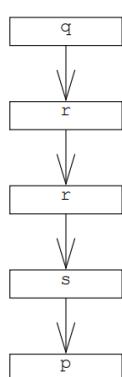
r: prima di terminare

s: prima di q

q: prima di terminare

s: prima di terminare

p: prima di terminare



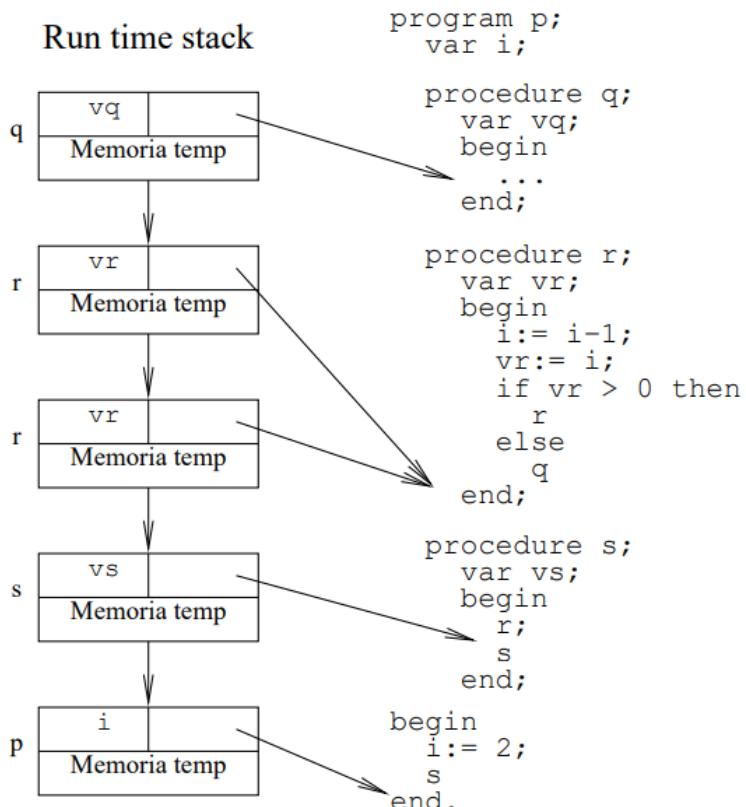
11. Record di attivazione

Ambiente locale

L'ambiente locale include:

- 1) Tutte le variabili dichiarate localmente
- 2) Puntatore alla prossima istruzione [IP] (i pallini rossi dell'esempio precedente); esso permette di riprendere l'esecuzione quando il controllo viene restituito alla procedura chiamante.
- 3) Memoria temporanea; necessaria alla valutazione delle espressioni contenute nella procedura (altamente dipendente dalla realizzazione, non esegue espressioni complesse ma suddivide quest'ultime in parti e ne salva i risultati parziali in questa memoria temporanea).

Esempio



12. Propagazione dei data object

Realizzazione

Viene realizzata aggiungendo al record di attivazione un puntatore al record di attivazione della procedura da cui vengono propagate le definizioni o i dati da cui ereditiamo le variabili non definite dalla procedura stessa. Se viene richiesto l'accesso ad un dato che non è definito localmente, esso viene ricercato in modo ricorsivo nei record di attivazione precedenti. Questa ricerca ricorsiva dipende dal tipo di scoping.

Sono definite tre tipologie di realizzazione della propagazione:

- 1) Propagazione in ambito statico. In questo caso l'ambiente non locale di una procedura è propagato dal programma che la contiene sintatticamente: propagazione di posizione.
- 2) Propagazione in ambito dinamico. In questo caso l'ambiente non locale di una procedura è propagato dal programma chiamante.
- 3) Nessuna propagazione. L'uso di ambienti non locali è scoraggiato perché produce **effetti collaterali** non facilmente prevedibili.

Ambito statico

Tutte le procedure innestate vedono le variabili non mascherate dei blocchi a loro esterni; le procedure possono chiamarsi tra di loro, e nello scoping statico le funzioni si comportano come le variabili, quindi un blocco interno può chiamare una procedura al suo esterno se non mascherata.

```

program p;
var a, b, c: integer;

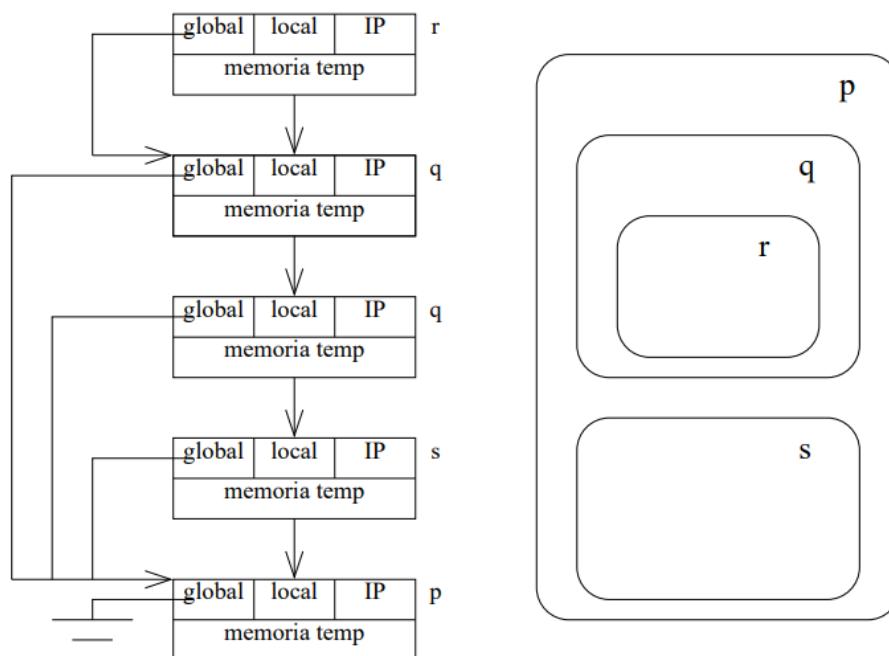
procedure q;
var a, c: integer;
procedure r;
var a: integer;
begin {r}           {variabili: a da r; b da p; c da q;
...                   procedure: q da p; r da q}
end; {r}
begin {q}           {variabili: a da q; b da p; c da q;
...                   procedure: q ed s da p; r da q}
end; {q}

procedure s;
var b: integer;
begin {s}           {variabili: a da p; b da s; c da p;
...                   procedure: q ed s da p}
end; {s}

begin {p}           {variabili: a, b, c da p;
...                   procedure: q, s da p}
end. {p}

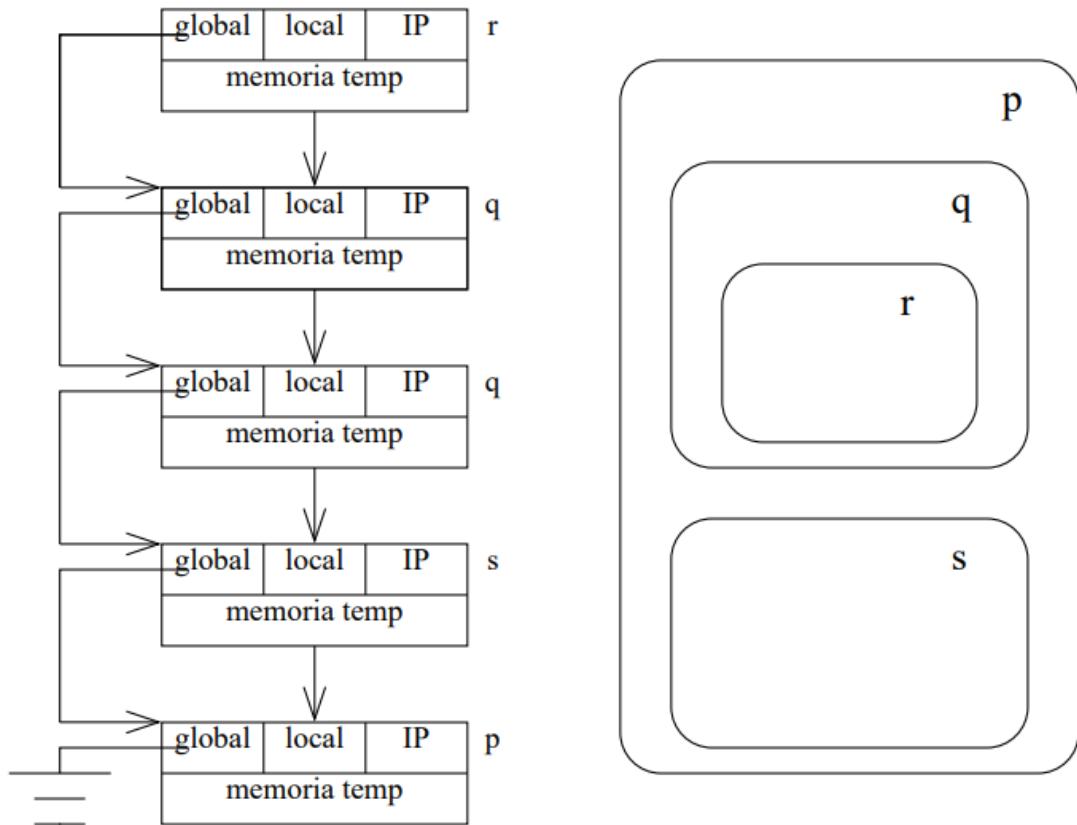
```

Supponendo una sequenza di attivazione dove *p* chiama *s*, *s* chiama *q*, *q* chiama ricorsivamente *q*, e infine *q* chiama *r*, lo stack di esecuzione ha questa forma:



Ambito dinamico

La stessa sequenza di attivazione descritta precedentemente (p, s, q, q, r), genera in ambito dinamico lo stack di esecuzione seguente (l'immagine di destra non rappresenta lo scoping dinamico ma serve a rimarcare la differenza con lo scoping statico):



Osservazioni

L'ambito statico dipende da come è stato scritto il programma, mentre l'ambito dinamico dipende da come è eseguito il programma, inoltre nella propagazione in ambito dinamico il puntatore all'ambiente non locale non è più necessario ed è praticamente impossibile la determinazione dell'ambiente di esecuzione di una procedura durante la scrittura del codice sorgente; per questo lo scoping dinamico è stato abbandonato.

```

program p;
  var a: integer;
  procedure q;
    begin          {vars: a da p o da r; procs: q da p}
    ...
  end;
  procedure r;
    var
      a: integer;
    begin          {vars: a da r; procs: q, r da p}
    ...
  end;
begin          {vars: a da p; procs: q, r, da p}
...
end.

```

13. Parametrizzazione di procedure

Parametri

I parametri sono la terza parte necessaria a specificare l'ambiente di esecuzione di una procedura e costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata. I parametri si possono distinguere in:

- Parametri IN: sono passati dalla unità chiamante alla unità chiamata al momento dell'invocazione;
- Parametri OUT: passati dalla unità chiamata alla unità chiamante al momento della terminazione della prima (utili per distribuire più di un valore, per un singolo valore si preferisce usare le funzioni);
- Parametri IN-OUT: servono a far transitare le informazioni in entrambe le direzioni.

I parametri devono essere specificati nella definizione della procedura (**parametri formali**) e nelle invocazioni della procedura (**parametri attuali**).

Associazione dei parametri

Regola comune:

- Nella definizione deve essere specificato il tipo dei parametri formali; nella invocazione è richiesta la corrispondenza di tipo tra parametri formali e attuali.

Eccezioni comuni:

- Lasciare i parametri formali senza alcun legame di tipo; il legame si instaura durante l'esecuzione (run time) allo stesso tipo dei parametri attuali (impossibile il type checking in compilazione).
- Permettere come eccezione solo quella degli array a dimensione variabile; il legame di tipo (e l'eventuale controllo di consistenza) verrà realizzato durante l'esecuzione.

Metodi di associazione dei parametri:

- Per **posizione** (attualmente usato): a seconda della posizione relativa nella sequenza dei parametri.
- Per **nome**: il nome del parametro formale è aggiunto come prefisso al parametro attuale.

Esempio:

- Data l'intestazione della seguente procedura in linguaggio ADA:
procedure TEST (A: in Atype; b: in out Btype; C: out Ctype)
- allora una invocazione che usa associazione per posizione è: **TEST(X, Y, Z);**
- mentre una che usa associazione per nome può essere: **TEST(A => X, C => Z, B => Y);**

Una ulteriore tecnica è la cosiddetta **associazione di default**. Essa permette di specificare valori di default ai parametri formali che non sono stati legati a valori da parametri attuali.

Parametri IN

Possono essere realizzati in due modi:

- 1) con un riferimento: in questo caso la locazione del parametro attuale diventa la locazione del parametro formale. Un parametro IN per riferimento non può essere modificato.
- 2) con una copia: in questo caso viene copiato il valore del parametro attuale in una nuova locazione, ovvero quella del parametro formale; in questo modo non ho più il vincolo di dover impedire la modifica all'interno della procedura poiché ho una copia di quell'elemento, ovviamente la copia può diventare onerosa nel caso di parametri lunghi.

Parametri OUT

Anche loro possono essere realizzati con un riferimento o con una copia. Questi parametri rappresentano risultati, di conseguenza alcuni linguaggi assumono che i parametri OUT non siano inizializzati e ne proibiscono la "lettura" (i.e. uso a destra di un assegnamento o passaggio a un parametro IN o IN OUT di un'altra procedura). Non esistono regole generali, ad esempio Ada 83 proibisce di "leggere" i parametri OUT, ma le versioni successive invece lo permettono. Praticamente ho un errore a tempo di compilazione se nella procedura non inizializzo le variabili prima di utilizzarle.

Parametri IN OUT

Sono la combinazione dei due precedenti, e come tale non ho restrizioni. Anch'essi possono essere realizzati:

- 1) con un riferimento; non ci sono limitazioni all'uso all'interno della procedura;
- 2) con una copia; avvengono due processi di copia, uno durante l'attivazione ed uno durante la terminazione della procedura (quando la procedura termina copia i valori nuovi nella chiamante).

Aliasing

È la possibilità di riferirsi alla stessa locazione con nomi diversi e questo può causare, nel passaggio dei parametri, notevoli problemi di interpretazione. Per esempio:

```
program MAIN;
var
  A: integer;
procedure TEST (var X, Y: integer);
begin
  X:= A + Y;
  writeln(A, X, Y)
end;
begin
  A:= 1;
  TEST(A, A)
end.
```

Esercizio: determinare l'uscita del programma nel caso in cui i parametri VAR siano realizzati per riferimento e nel caso in cui siano realizzati per copia.

SOLUZIONE: Riferimento: $A = 2, X = 2, Y = 2$ essendo $X, Y = \text{env}(A)$ Copia $A = 1, X = 2, Y = 1$,
inoltre possiamo osservare che l'unica modalità di riferimento accettata è IN-OUT, poiché X sarebbe
proibita dalla modalità IN e Y dalla modalità OUT

Procedure come parametri di procedura

Alcuni linguaggi permettono l'uso di procedure come argomento di altre procedure. Esempio:

```
program MAIN;
  VAR a: real;
  procedure TESTPOS (X: real; procedure ERROR (MSG: string));
  begin
    if X <= 0 then ERROR ('Negative X in TESTPOS')
  end;
  procedure E1 (M: string);
  begin
    writeln('E1 error: ', M)
  end;
  procedure E2 (M: string);
  begin
    writeln('E2 error: ', M)
  end;
begin
  readln (A);
  TESTPOS(A, E1);
  TESTPOS(A, E2)
end.
```

Macro

Generazione di un nuovo brano di codice sorgente (espansione della macro) in cui i nomi dei parametri attuali sostituiscono i nomi dei parametri formali.

Le macro in C, ad esempio, sono delle linee di codice tipo `#define COST 1`, in questo modo tutto le variabili COST nel codice viene rimpiazzato con 1 dal preprocessore. È possibile anche utilizzare le macro per le funzioni: `#define MACRO(x) printf(x)` (si usano di rado poiché è un'operazione molto rischiosa).

Esempio: data la procedura

```
procedure swap (a, b: integer);
var temp: integer;
begin
  temp := a;
  a := b;
  b := temp
end;
```

allora la chiamata `swap(x,y)` esegue il seguente brano di codice:

```
temp := x;
x := y;
y := temp;
```

Esercizio: Determinare i problemi che nascono dai due programmi, se swap è una macro:

```
program main;
var
  i: integer;
  m: array[1..100] of integer;
  ...
begin
  ...
  swap(i, m[i]);
  ...
end.
```

SOLUZIONE: Con `swap(i, m[i])` avremo in successione `temp := i; i := m[i]; m[i] := temp` (vedi la chiamata swap descritta sopra nell'esempio) risulterà quindi che i due `m[i]` non siano la stessa cosa ma siano copiati dentro i parametri formali evitando cambiamenti dinamici) due celle diverse del vettore, poiché non c'è un record di attivazione (un posto dove i parametri attuali

Funzioni

Sono procedure che restituiscono un valore alla procedura chiamante. Sono realizzate o creando una pseudovariabile nell'ambiente locale della procedura chiamata (tale variabile può essere solo modificata; non è possibile l'accesso in lettura); o utilizzando una istruzione di return per restituire esplicitamente il controllo alla procedura chiamante inviandole allo stesso tempo il valore di una espressione.

14. Implementazione efficiente dell'ambiente non locale con scoping statico

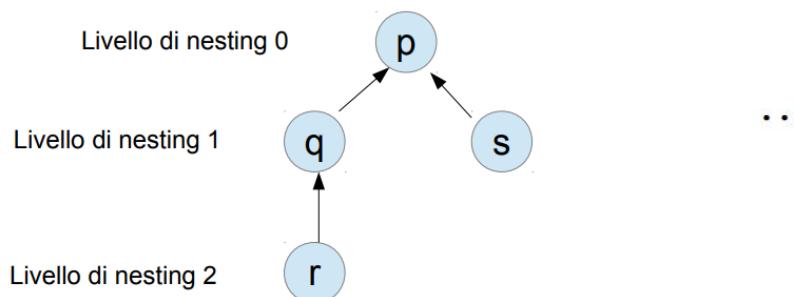
Annidamento (nesting)

Una procedura q è **annidata** in un blocco b se è definita dentro b , ad esempio, in figura, q e s sono annidate in p mentre r è annidata in q .

Il **livello di nesting** di una procedura è il numero di blocchi che la contengono: il livello di nesting di q e s è 1 mentre quello di r è 2

L'**ambiente statico non locale** di una procedura è dato dall'ambiente delle procedure in cui è innestata.

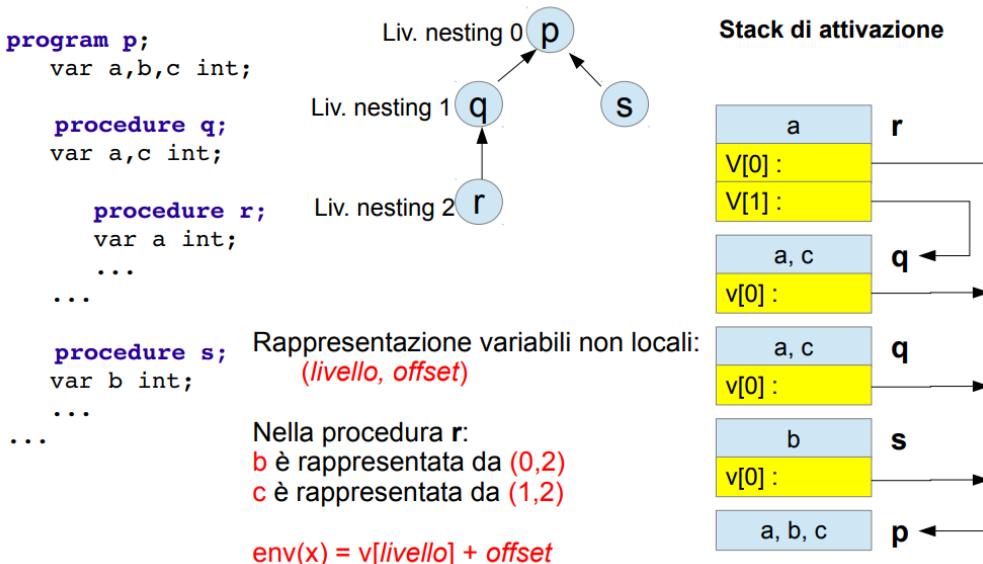
L'annidamento si può rappresentare come un albero dove i livelli dell'albero corrispondono ai livelli di nesting:



Le frecce indicano l'ambiente non locale

Rappresentazione variabili non locali

Posso rappresentare le variabili non locali con la coppia (*livello, offset*) dove ho, rispettivamente, le informazioni del livello di nesting (blocco dove si trova la variabile) e della posizione nel record di attivazione (posizione in cui viene dichiarata la variabile). Quindi il nostro ambiente sarà un puntatore sul livello e l'offset: $env(x) = v[livello] + offset$ (questa operazione di differenziare un puntatore aggiungendo l'offset è così comune che è possibile farlo con una singola operazione macchina).



Visibilità delle procedure

Siano F e G due procedure, F può chiamare G se: G è definita in F oppure se G è definita in uno dei blocchi che contiene F ; quindi F e G hanno sempre una parte di ambiente non locale in comune.

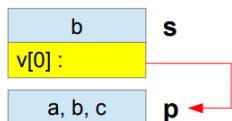
Mantenimento del vettore degli ambienti non locali

Di seguito rappresentiamo esempi di chiamate a procedure:

- definite **localmente**

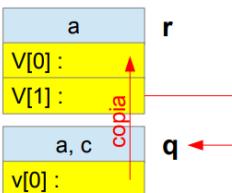
- *p* chiama *s*:

- Aumenta livello di nesting
 - Aggiungere elemento a $v[]$



- o *q* chiama *r*:

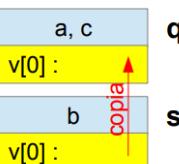
- Aumenta livello di nesting
 - Aggiungere elemento a $v[]$
 - Copiare gli altri elementi



- definite esternamente

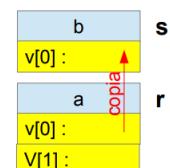
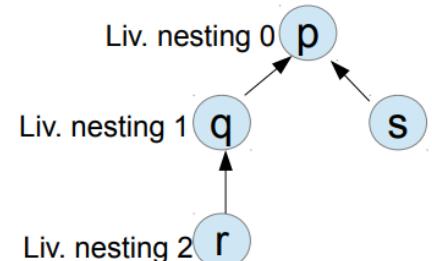
- *s* chiama *q*:

- Stesso livello di nesting
 - Copiare gli elementi di $v[]$



- o r chiama s:

- il livello di nesting **decresce**
 - Copiare solo gli elementi di livello *minore o uguale* a quello di *s*
 - Struttura ad albero + scoping statico garantiscono sempre di trovare l'ambiente non locale della procedura chiamata nel record del chiamante



Proprietà di questa implementazione

- Accesso alle variabili in tempo costante
 - Accesso a vettore + somma del puntatore ivi contenuto e dell'offset
 - Indipendente dai livelli di nesting
 - Calcolo supportato direttamente dalle istruzioni macchina
 - Creazione record di attivazione lineare nel livello di nesting (copia degli elementi di $v[]$)
 - Indipendente dall'esecuzione
 - Dipende solo dal testo del sorgente
 - Tempo costante
 - L'implementazione naïve richiederebbe a run time di percorrere le liste di puntatori all'ambiente non locale
 - Accesso alle variabili in tempo lineare nel livello di nesting
 - Creazione dei record di attivazione in tempo lineare nel livello di nesting (istantaneo)

Analisi e Progettazione O.O.; UML

15. Modello orientato ad oggetti

Introduzione

Scrivere un programma in un linguaggio OO (Object Oriented) non significa scrivere un programma OO, infatti richiede prima un processo di OOAD (Object Oriented Analysis and Design).

La progettazione OO ha i seguenti vantaggi:

- 1) Progettazione elegante e facile da capire.
- 2) Programmi eleganti e facili da capire.
- 3) Oggetti singoli realizzati e mantenuti in modo indipendente.
- 4) Librerie di oggetti facilmente riutilizzabili e riadattabili ad un nuovo progetto.
- 5) Facilità di modifica e di debug.

Tra i linguaggi OO sono molti (Smalltalk, Eiffel, Ada, ...) ma il mercato ha decretato come vincenti C++ e Java:

- Perché C++?
Acquisisce eredità di C ed è compatibile con programmi C esistenti. È tuttavia cresciuto in modo abnorme ed è veramente difficile raggiungere piena competenza in tutti gli aspetti del linguaggio.
- Perché Java?
WWW e abilità di eseguire “web applet” su qualunque elaboratore e sistema operativo, attraverso un browser web. Linguaggio ben progettato, con pochi costrutti. Risolti i problemi della velocità di esecuzione. Diffusamente utilizzato nell’ambito educativo (prima volta che un linguaggio di produzione reale coincide con un linguaggio di insegnamento).

Oggetto

Un oggetto è la rappresentazione di ogni cosa in un programma OO; può essere il modello di un sensore, una finestra in una interfaccia utente, una struttura dati, ..., virtualmente tutto. Possiamo vedere un oggetto come una scatola nera con pulsanti e spie dove per utilizzarlo basta sapere solo a cosa servono i pulsanti (quale premere per ottenere ciò che si vuole) ed il significato dei segnali luminosi delle spie. Ciò che importa è che l’oggetto esegua correttamente le proprie funzioni e si faccia carico delle proprie responsabilità. L’interno della scatola è completamente nascosto all’ambiente esterno. In essenza: Progettare un sistema OO consiste nell’identificare quali oggetti il sistema debba contenere, il comportamento e le responsabilità di ognuno di essi e come essi interagiscono tra loro.

Essenza di un oggetto:

- Insieme di attributi (valore, stato interno, o qualunque altra cosa sia necessaria per il modello dell’oggetto).
- Abilità di modificare lo stato degli attributi.
- Responsabilità sotto forma di servizi offerti ad altri oggetti. Gli oggetti esterni non hanno conoscenza di come un oggetto realizza gli attributi internamente, ma piuttosto devono conoscere quali servizi un oggetto offre.

Metodologie OOAD e UML

Le metodologie Object Oriented Analysis and Design sono numerose e classificate in: **Heavyweight** (le più datate e tradizionali, usate per grandi progetti software che coinvolgono centinaia di programmatore in anni di lavoro) e **Lightweight** (agile; per progetti più piccoli). Ogni metodologia sviluppava una propria notazione grafica ma tutto ciò è cambiato con l’introduzione di un UML (Unified Modelling Language); progettato per discutere su OOAD e per descrivere progetti esistenti. L’UML può essere anche visto come un vero e proprio linguaggio di programmazione i cui simboli sono diagrammi “ben formati”.

UML è utilizzato sia in fase di **concettualizzazione** (quando si devono esplicitare le specifiche di funzionamento di un sistema) sia in fase di **analisi software** (quando si devono descrivere i funzionamenti di un progetto software esistente) e sia in fase di **sintesi software** (quando si deve progettare un sistema software, ed in questo caso UML è un vero e proprio linguaggio di programmazione).

Classe

Una classe è la descrizione di un insieme di oggetti che condividono attributi e comportamento comuni. Il concetto di classe è simile a quello del tipo di dato astratto nei linguaggi non OO.

La definizione di una classe descrive tutti gli attributi comuni agli oggetti della classe, così come tutti i metodi che realizzano il comportamento comune degli oggetti della classe. Gli oggetti membri di una classe sono chiamati **istanze** di quella classe. I membri di una classe variano durante l'esecuzione di un programma.

Esempi:

- 1) In un modello di un ROBOT un tipo di oggetto è ovviamente il sensore. La classe *Sensore* definisce le caratteristiche di base di un qualunque sensore, come la posizione nello spazio, il valore del trasduttore, un codice identificativo ed un insieme di servizi offerti per adempiere alle proprie responsabilità, i mezzi per accedere e modificare lo stato interno degli oggetti individuati dalla classe. Ogni oggetto istanza di *Sensore* avrà valori specifici per gli attributi descritti nella definizione di classe.
- 2) Una rappresentazione comune del colore è la RGB, dove ogni colore è specificato dalle componenti R (Red), G (Green), B (Blu). Una possibile definizione di una classe chiamata *Colore* dovrebbe fornire i mezzi per accedere o modificare il colore di un oggetto *Colore*.

Gerarchie

In un sistema ad oggetti è tipico definire una classe basandosi su classi preesistenti o estendendo la descrizione in una classe di più alto livello oppure includendo la descrizione di un'altra classe all'interno della classe corrente. Si possono, per esempio, costruire le classi che descrivono sensori di pressione o di temperatura basandosi sulla generica classe *Sensore*.

Aggiungere attributi e metodi → sottoclasse
Sottrarre attributi e metodi → superclasse

Vantaggi: se la stessa responsabilità è condivisa da più di una sottoclasse, porre il metodo che la realizza nella superclasse riduce le ripetizioni.

16. Peculiarità della programmazione OO

Astrazione con oggetti

L'**astrazione** è il meccanismo che permette di rappresentare una situazione complessa del mondo reale con un modello semplificato. Quella OO astrae il mondo reale usando oggetti e la loro interazione.

Un tipo di dato astratto è realizzato nei linguaggi di programmazione tramite **encapsulation**; encapsulation significa che le strutture dati che implementano i valori del tipo di dato sono accessibili direttamente solo dalle operazioni fornite dal tipo (perfetta encapsulation con i metodi private). Questa è una regola di scoping.

Classi incapsulate

Le classi incapsulate nascondono la realizzazione di attributi e del comportamento di un oggetto al resto del mondo e permettono a ciascun oggetto di essere indipendente, inoltre vengono realizzate pensando alle proprietà degli oggetti che dovranno rappresentare:

- **Attributi**: possono essere strutture semplici, come una variabile, oppure complesse, come un altro oggetto.
- **Comportamento**: attività dell'oggetto così come viene vista all'esterno.

- **Metodi:** realizzano il comportamento dell'oggetto; sono operazioni o servizi offerti dall'oggetto all'esterno.
 - **Stato:** riflette i valori correnti degli attributi che caratterizzano l'oggetto

Scambio di messaggi

Lo scambio di messaggi è il metodo con cui gli oggetti interagiscono; praticamente un oggetto invia messaggi ad un altro per richiedere informazioni sullo stato interno o richiedere modifiche dello stato interno. I messaggi possono essere:

- **Sincroni:** nei sistemi che simulano operazioni effettive oppure quando la risposta è necessaria per la continuazione del processo associato all'oggetto mittente.
 - **Asincroni:** l'oggetto mittente non attende la risposta, ma il suo processo prosegue in modo indipendente al processo del destinatario.

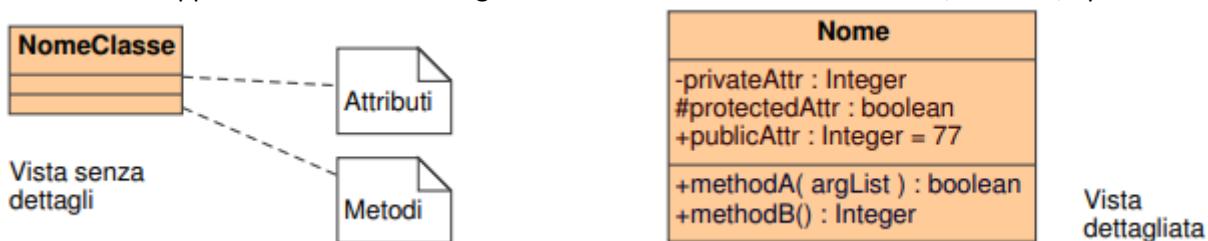
Ciclo di vita di oggetti

Gli oggetti sono entità dinamiche quindi nascono durante l'esecuzione del programma come istanze di una certa classe, evolvono e poi muoiono. Alla generazione di una istanza, viene invocato un particolare metodo della classe genitrice chiamato **costruttore**. Esso è responsabile dello stato iniziale dell'istanza mentre alla morte di un oggetto alcuni linguaggi richiedono l'invocazione esplicita di un metodo (dell'istanza) chiamato **distruttore**; altri linguaggi possiedono un meccanismo implicito di distruzione: un oggetto cessa di esistere quando non vi sono più riferimenti ad esso; a questo punto è il sistema stesso che riconosce i "cadaveri" e li elimina (**garbage collection**).

17. Elementi di UML

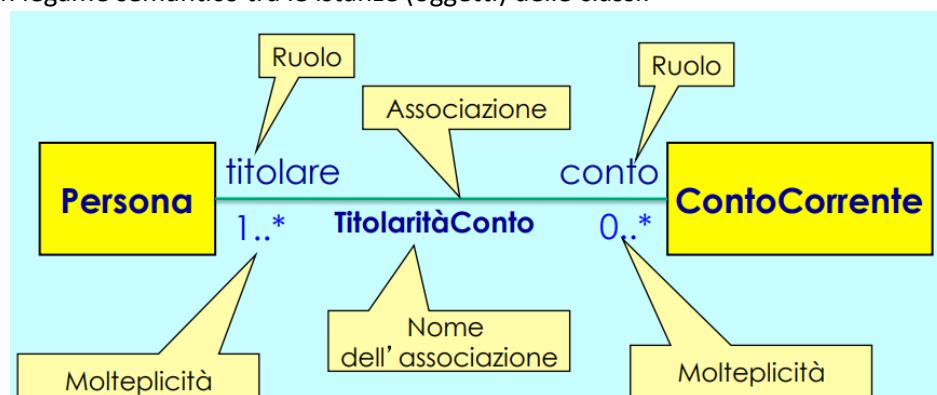
Diagrammi di classe

Una classe è rappresentata da un rettangolo con tre diverse sottosezioni: nome, attributi, operazioni.



Associazione

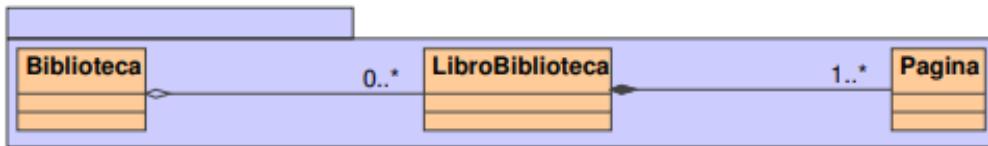
Attributo particolare in una classe, rappresentato in UML da una linea continua. L'associazione tra due (o più) classi esprime un legame semantico tra le istanze (oggetti) delle classi:



La molteplicità vincola il numero di oggetti di una classe che possono partecipare ad una relazione in un dato istante.

Composizioni e Aggregazioni

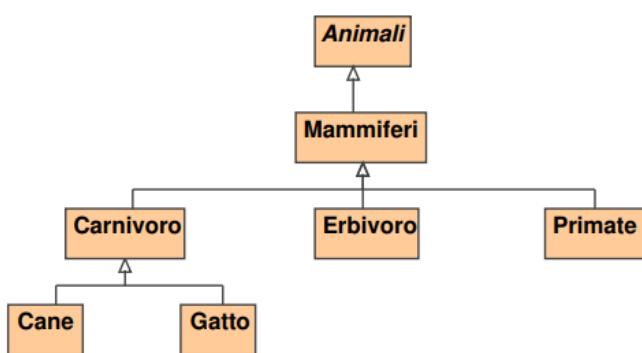
Relazione "ha_un"



- **Composizione:** oggetto composto esiste finché esistono le componenti (rombo pieno); inoltre, se non esiste più l'oggetto composto, non esisteranno più nemmeno le componenti.
- **Aggregazione:** oggetto aggregato esiste anche se non esistono gli aggreganti (rombo vuoto); inoltre, anche se non esiste più l'oggetto aggregato, gli aggreganti restano in vita.

Generalizzazioni

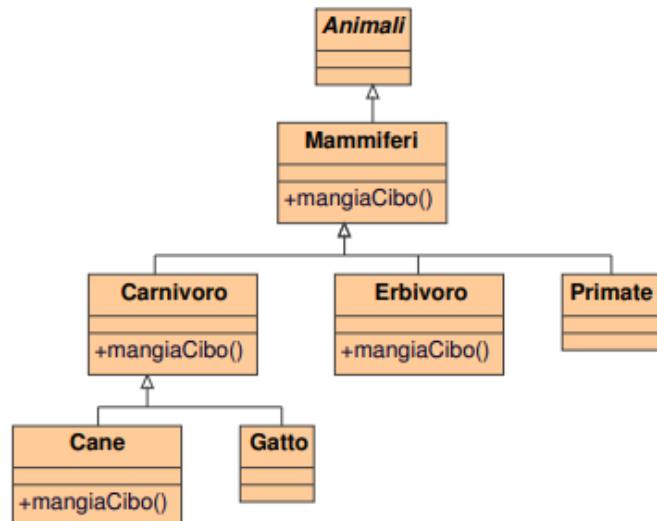
Relazione "è_un"



È realizzata mediante ereditarietà. Ogni sottoclasse è una specializzazione della superclasse ed eredita da essa (solo alcuni) attributi e/o metodi.

Polimorfismo

Permette al mittente di ignorare l'identità (il tipo) del ricevente. È realizzato dal mix di overriding e upcasting.



Il polimorfismo per inclusione è basato sull'overriding (ho più metodi con lo stesso nome ma implementazioni diverse) e linking dinamico dei metodi (l'associazione tra il metodo da seguire e la classe in cui è implementata è fatta run time).

18. Altri concetti OO

Classi astratte

Definiscono una interfaccia comune a tutte le sottoclassi, specificando solo i nomi (la firma) dei metodi che le sottoclassi "concrete" dovranno definire ma non la loro implementazione. Per esempio: tutti gli Animali devono avere un metodo riproduci, ma ogni sottoclasse definirà poi il metodo più appropriato.

Poiché sono incomplete della realizzazione dei metodi, le classi astratte non possono avere istanze. Nel linguaggio UML sono scritte in *corsivo*.

Visibilità delle componenti di una classe

È la possibilità di una classe di vedere ed usare le risorse di un'altra classe.

Livelli di visibilità:

- **Private:** attributi e metodi sono visibili solo a oggetti istanze della stessa classe. *In UML: prefisso -*
- **Package:** attributi e metodi sono visibili ad un insieme specifico di classi. *In UML: prefisso ~*
- **Protected:** oltre alla precedente, attributi e metodi sono visibili a tutti gli oggetti istanze della stessa classe o di una sua sottoclasse. *In UML: prefisso #*
- **Public:** Attributi e metodi sono visibili a tutti gli altri oggetti. *In UML: prefisso +*

Normalmente gli attributi non sono mai pubblici. Solo alcune operazioni sono pubbliche, quelle che forniscono servizi. Se si devono reperire o modificare le informazioni sullo stato interno di un oggetto, lo si fa utilizzando i metodi specifici (e visibili) dell'oggetto.

19. Diagrammi UML

UML: richiami

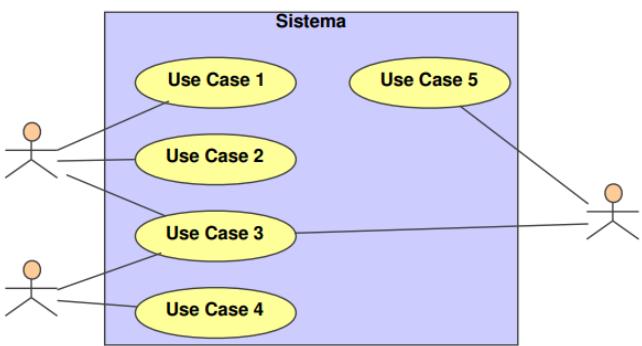
L'UML è un linguaggio grafico per specificare, costruire, visualizzare e documentare; nell'UML è la notazione che descrive il ciclo completo dello sviluppo OO. Ci sono diverse scritture degli stessi diagrammi, con maggiore o minore dettaglio, a seconda dell'ambito di lettura e della prospettiva scelta per visualizzare il sistema.

Modello statico

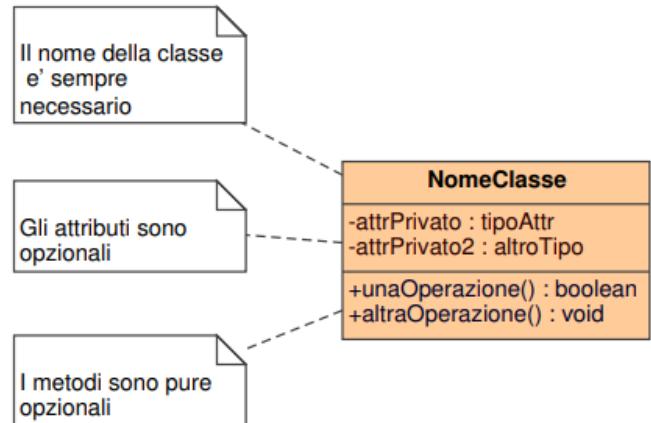
Il modello statico rappresenta la struttura base del sistema software che si sta modellando; questo modello crea una rappresentazione degli elementi principali del dominio del problema. Praticamente costruisce e documenta l'aspetto statico del sistema.

Comprende i seguenti diagrammi:

- **Use Case Diagram:** utili per identificare le proprietà di base o le specifiche richieste al sistema.
 - Mette in evidenza le specifiche del sistema.
 - Mostra chi o cosa usa il sistema.
 - Gli utenti di una certa funzionalità sono detti attori.
 - I casi d'uso sono rappresentati con ellissi.
- **Class Diagram:** sono i più usati; essi forniscono diverse visualizzazioni del sistema, sono uno strumento per esplorare il vocabolario del dominio del problema, sono un modo per documentare successive realizzazioni.

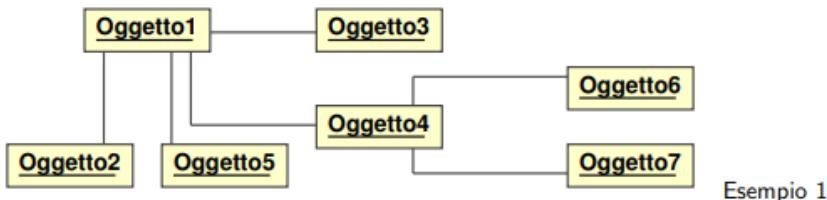


- Diagrammi concettuali: usati spesso dagli analisti per meglio individuare i concetti ed il vocabolario del dominio del problema. In questa prospettiva bisogna escludere i dettagli realizzativi specifici del linguaggio.
- Diagrammi realizzativi: contenenti maggiori dettagli riguardanti metodi ed attributi necessari per realizzare le classi



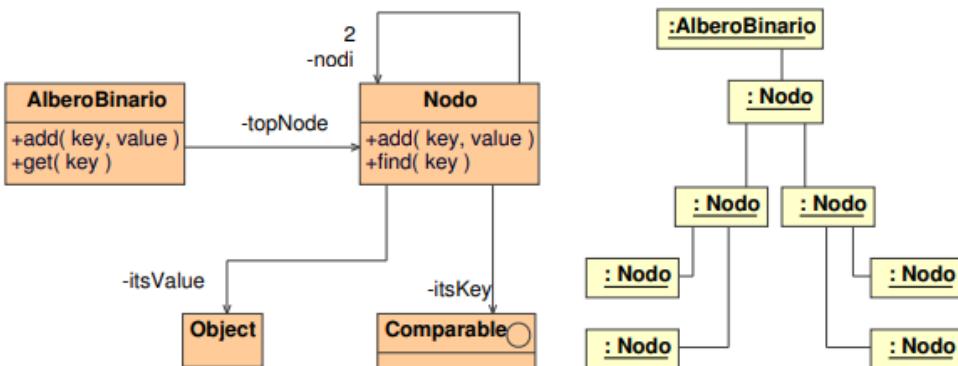
- **Object Diagram**: aiutano a identificare l'organizzazione strutturale tra oggetti.

- È un diagramma usato per rappresentare oggetti specifici che esistono ad un certo istante durante il ciclo di vita del sistema.



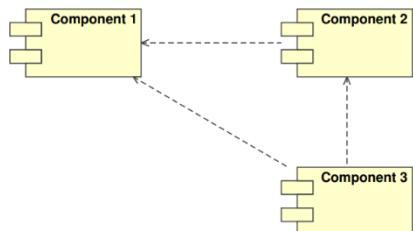
Esempio 1

Esempio 2 (dalle classi agli oggetti):



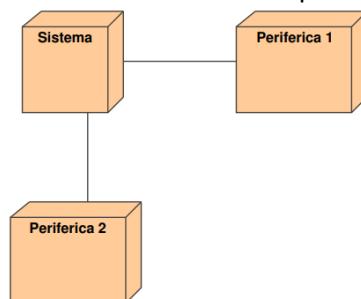
- **Component Diagram**: aiutano a localizzare applicazioni software.

- Mostrano le relazioni tra componenti software.
- Le dipendenze sono rappresentate da linee tratteggiate.



- **Deployment Diagram**: specificano la mappa hardware.

- Rappresenta i dispositivi fisici che saranno usati per eseguire le applicazioni software

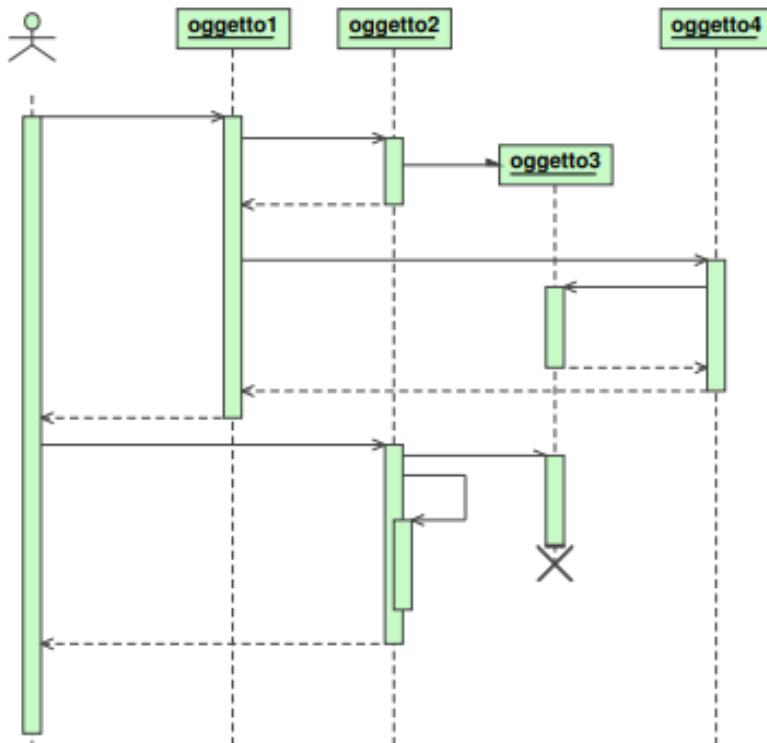


Modello dinamico

Riguarda il comportamento del sistema. Comprende i diagrammi:

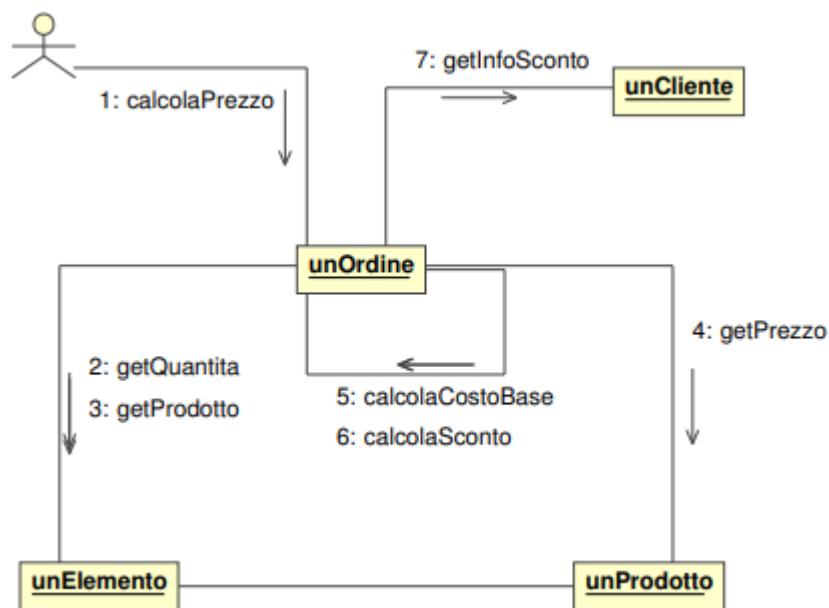
- **Sequence Diagram:** mostrano la sequenza temporale di messaggi durante una particolare attività del sistema.

- Cattura i messaggi che sono scambiati in un periodo di tempo tra diversi oggetti.
- Pone l'accento sull'ordine di successione temporale dei messaggi.

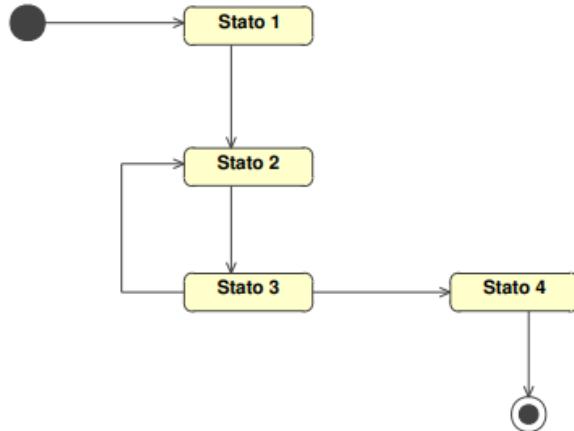


- **Collaboration Diagram:** mostrano messaggi tra oggetti e la struttura tra gli oggetti.

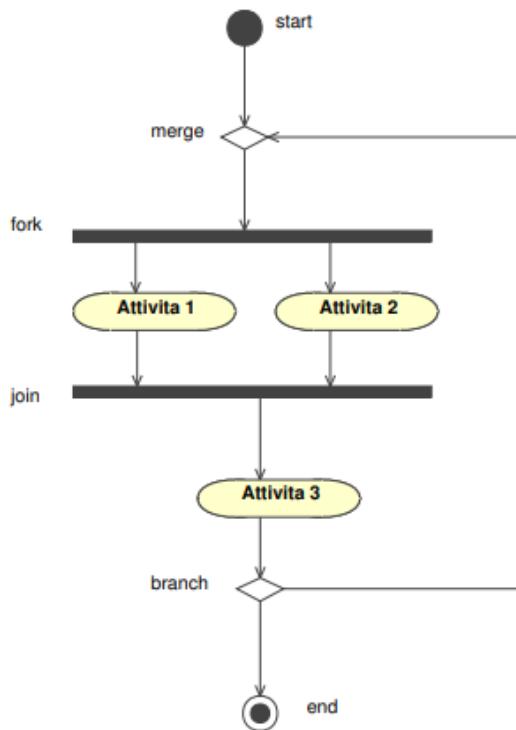
- Mostra la collaborazione di oggetti mediante messaggi.
- I messaggi possono essere numerati a vari livelli.
- Pone l'accento sull'organizzazione strutturale degli oggetti che scambiano messaggi.



- **State Diagram:** esaminano gli stati interni e le transizioni tra stati di un singolo oggetto.
 - Mostra una rappresentazione di un singolo oggetto come automa a stati finiti.
 - Pone l'accento sul comportamento di un singolo oggetto in risposta ad eventi esterni.



- **Activity Diagram:** esaminano il comportamento di più oggetti.
 - Descrivono il flusso da una attività alla successiva



20. Fase di analisi iniziale

Attività

Durante la fase di analisi iniziale: viene individuato il problema; sono analizzati i flussi di lavoro di ciascuna iterazione nel ciclo di sviluppo; sono raccolte le informazioni; viene creato l'enunciato del problema; sono generati i diagrammi Use Case.

Ovviamente bisogna evitare pregiudizi come “Gli utenti sono ingenui, gli sviluppatori no”, “Le richieste o le specifiche sono statiche”, e per farlo bisogna ricordare che il sistema deve fornire le funzionalità richieste (i progetti evolvono costantemente e le richieste cambiano); si devono identificare chiaramente le richieste dell’utente; bisogna assicurarsi che il modello possa adattarsi a richieste in evoluzione; ed anche prevedere di poter correggere il modello se vi saranno imprecisioni di informazioni nel dominio del problema.

Reperire info

Si possono reperire informazioni attraverso questionari o gruppi di discussione da numerose fonti:

- **Richieste iniziali del cliente:** di solito un documento non sempre preciso; bisogna intervistare le persone giuste, riempire vuoti, capire in anticipo la tecnologia necessaria.
- **Esperti del dominio:** nel settore di attività del cliente, di solito analisti presentati dal cliente. Sono gli specialisti in un'area particolare relativa al problema.
- **Utenti finali:** sono coloro che forniranno informazioni pratiche sull'uso del prodotto, contro le informazioni teoriche precedenti.
- **Dirigenti:** sono coloro che dovranno gestire il prodotto.
- **Mercato:** deve essere esplorato per riconoscere se esiste una richiesta per prodotti simili e nel caso considerare di costruire un modulo o componenti generiche per promuovere il riuso in progetti simili;
- **Precedenti progetti:** forniscono informazioni (positive o negative) su altre esperienze.

Enunciato del problema

È il documento che descrive chiaramente le richieste del cliente. Esso deve dichiarare: tutte le informazioni che sono pertinenti all'analisi e al disegno del progetto; tutti i vincoli esistenti che il progetto di sviluppo del sistema deve tenere in considerazione; i flussi di informazione attraverso il sistema; gli utenti finali del prodotto; l'ingresso e l'uscita del sistema.

Deve essere scritto in un linguaggio specifico del dominio del problema. Deve contenere frasi complete, senza abbreviazioni, e nessuna o poca terminologia di informatica.

L'enunciato del problema può evolvere ed è usato come base per individuare il dominio del problema, inoltre nell'enunciato si identificano una possibile lista di oggetti e classi che il sistema deve contenere (sottolineare sostantivi dall'enunciato del problema per costruire la lista degli oggetti e delle classi candidate).

Dominio del problema

È l'enunciato, grafico o in forma di testo, che descrive quali aree e problemi dovranno essere di pertinenza del sistema. Il cliente non è coinvolto a questo livello ma dovrà essere coinvolto durante il progetto, per chiarire e convalidare tutte le aree del dominio del problema.

21. Analisi di oggetti e classi

Introduzione

La fase di analisi identifica gli oggetti richiesti in esecuzione per assicurare la funzionalità del sistema. Viene dopo la fase di *individuazione delle specifiche* e degli *Use Case* e prima della fase di *progetto del sistema*.

Durante la fase di analisi si definisce cosa deve fare il sistema evitando di descrivere dettagli di progettazione e realizzazione; pone l'accento sui componenti del sistema. Durante questa fase bisognerebbe rispondere alle seguenti domande:

- Quali sono gli oggetti del sistema?
- Quali sono le possibili classi?
- Come sono correlati gli oggetti?
- Quali sono le responsabilità di ciascun oggetto o classe?
- Come sono correlati oggetti e classi?

Astrazioni chiave

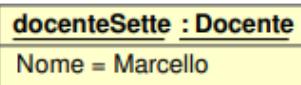
Riferiscono ad una sottolista di parole all'interno della lista dei possibili oggetti. Rappresentano gli oggetti primari o principali nel sistema. Sono identificate dopo aver studiato ciascun possibile oggetto e aver deciso se è abbastanza importante per essere una **Astrazione Chiave**.

Oggetti e Classi

Il passo successivo è quello di rappresentare gli aspetti logici e fisici del modello statico. I diagrammi utilizzati sono i **Class Diagram** (mostrano le classi che dovrebbero costituire il sistema, insieme con tutte le possibili relazioni tra esse) e gli **Object Diagram** (rappresentano gli oggetti presenti nel sistema ad un certo istante di tempo). Entrambi i diagrammi utilizzano la stessa sintassi di base e possono essere prodotti contemporaneamente. Di solito i diagrammi Class sono prodotti prima di quelli Object (pensare prima in modo astratto, poi specificare e verificare), anche se alcuni analisti preferiscono lavorare in modo inverso (prima elencare i particolari, poi produrre le astrazioni).

Classi e oggetti in UML:

- Gli oggetti non hanno la sezione dei metodi, poiché essi possiedono tutti i metodi della classe di cui sono istanze.
- Analogamente gli oggetti non hanno la sezione degli attributi, ma possono avere una sezione in cui sono elencati solo i valori degli attributi della classe che rendono unico quell'oggetto.



Attributi e metodi

Non sono noti prima della fase di Progettazione ma alcuni sono già ovvi e possono essere aggiunti anche nella fase di Analisi; in questa fase il fatto che un attributo o un metodo esista è già sufficiente, per questo non è essenziale aggiungere informazioni sul tipo di un attributo o sui parametri di un metodo.

Gli attributi che si possono identificare nella fase di analisi sono i nomi del dizionario dati che non erano stati classificati come astrazioni chiave, poiché il fatto che essi siano stati utilizzati per descrivere il sistema, significa che sono rilevanti in qualche modo.

22. Relazione tra classi

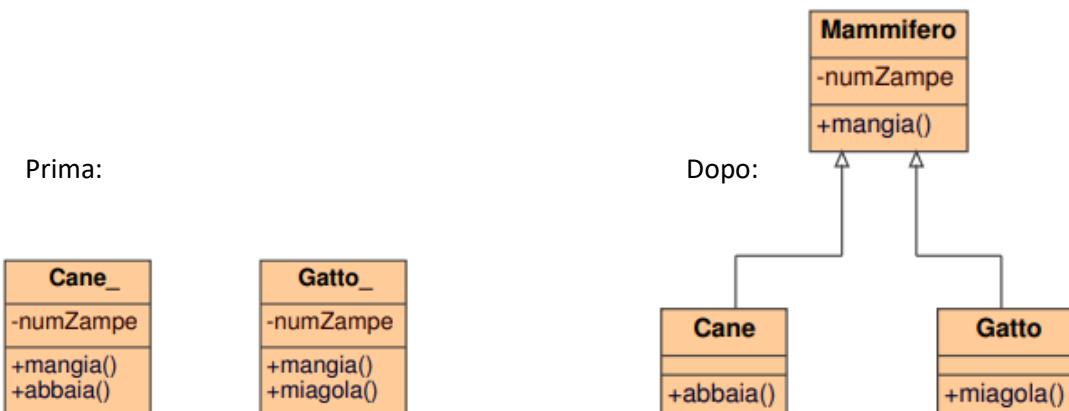
Ereditarietà

Il concetto di ereditarietà descrive come si possano condividere attributi e funzionalità tra classi di natura o scopo simile. L'ereditarietà può essere singola: ogni classe ha un solo genitore (eccetto la radice che non ne ha nessuno); o multipla: una classe può avere più di un genitore. Inoltre, con **superclasse** si intende la classe più "generale" (quella preesistente), mentre con **sottoclasse** la classe più "specializzata", quest'ultima eredita dalla superclasse i membri preesistenti ed estende la superclasse con nuovi membri.

Vi sono due modi con cui si può aggiungere il concetto di ereditarietà al modello del sistema:

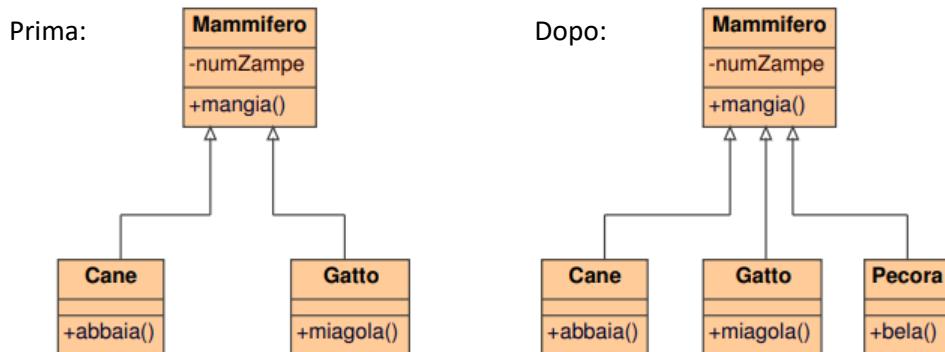
- **Generalizzazione**

- Avviene quando si riconosce che più classi dello stesso Class Diagram esibiscono funzionalità, struttura, scopo comuni.
- L'analista decide allora di creare una nuova classe contenente gli attributi e le funzionalità comuni e semplifica le classi precedenti come estensioni della nuova (generalizzata).



- **Specializzazione**

- Approccio opposto alla generalizzazione, avviene quando si riconosce che la nuova classe che si sta aggiungendo ha le stesse funzionalità, struttura e scopo di una classe esistente, ma ha bisogno di nuovo codice o attributi. La nuova classe è una forma specializzata di quella già esistente



Polimorfismo

Il polimorfismo è strettamente correlato con l'ereditarietà, prendendo gli esempi precedenti si può dire che vi sono molte forme di *Mammiferi* nel sistema. L'effetto pratico è che un riferimento ad un *Mammifero* può essere usato sia per riferirsi ad un oggetto *Cane*, sia ad un oggetto *Gatto*. Senza polimorfismo sarebbe difficile scrivere metodi che possono operare su diversi tipi di oggetti. Ad esempio, in una classe si può dichiarare una variabile per riferirsi ad un *Mammifero* prima che si sappia esattamente quante classi estendano *Mammifero* e quali saranno istanziate dal sistema in una particolare circostanza. Praticamente il polimorfismo (per inclusione) si riferisce al fatto che un'espressione il cui tipo sia descritto da una classe *A* può assumere valori di un qualunque tipo descritto da una classe *B* sottoclasse di *A*.

Classi astratte

Contengono funzionalità incomplete (metodi astratti, privi di realizzazione) e non può avere istanze. Una classe che estende una classe astratta eredita tutti i suoi metodi (inclusi quelli astratti). Ogni classe che eredita metodi astratti è considerata astratta essa stessa, a meno che essa non realizzi tutti i metodi astratti ereditati, in pratica se una classe ha un metodo incompleto (chiamato metodo astratto) allora è astratta, viceversa, non è detto che una classe dichiarata astratta abbia metodi incompleti.

Associazioni e molteplicità

L'associazione tra due (o più) classi esprime un legame semantico tra le istanze (oggetti) delle classi; una associazione è caratterizzata da:

- un **nome**: esprime il legame semantico tra le classi associate
- un eventuale **ruolo** giocato da ciascuna delle parti associate
- la **molteplicità** dell'associazione: vincola il numero di oggetti di una classe che possono partecipare ad una relazione in un dato istante, se la molteplicità non è indicata esplicitamente, essa è indefinita
Sono della forma *minimo..massimo*
 - 0..1 Zero (partecipazione opzionale) o uno
 - 1 Esattamente 1
 - 0..* Zero o più
 - * Zero o più
 - 1..* Uno (partecipazione obbligatoria) o più
 - 1..6 Da 1 a 6
 - 1..4,7,9 Da 1 a 4, oppure 7, oppure 9
- la **navigabilità** dell'associazione

Associazioni complesse

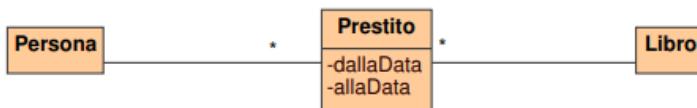
Sono quelle in cui vi sono molteplicità maggiori di 1 in entrambi i ruoli.



È difficile pensare ad una possibile realizzazione, e per la risoluzione ci sono due tecniche:

- **Classe di associazioni**

- Il nome della classe deve essere lo stesso della associazione.
- Una classe di associazione è usata per documentare la risoluzione di una associazione complessa, non fa parte delle astrazioni chiave.
- Restano nei Class Diagram finché, nella fase di progettazione, non si decide di realizzarle.
- Per realizzarle, bisognerà inserirle in altro modo nei diagrammi:



- **Associazione qualificata**

- Usate quando si vuole evidenziare che la realizzazione futura sarà mediante un array, una hash table, un dizionario, etc...
- In questo caso, se, per esempio, nella biblioteca ogni lettore (istanza di *Lettore*) ha il proprio unico codice identificativo (indice nell'array), ogni accesso dal punto di vista del *Libro* avviene mediante tale codice identificativo:



Viceversa, se nella biblioteca ogni libro è registrato usando un unico codice, il riferimento ad esso, dal punto di vista del Lettore, avviene mediante tale codice:

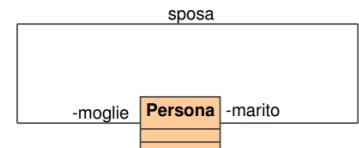


- La differenza realizzativa è che, mentre con una vera e propria classe di associazioni, le informazioni sul singolo prestito sono localizzate in una istanza di quella classe, nel caso delle associazioni qualificate, le informazioni sui prestiti sono delocalizzate e modificare un prestito significa modificare una istanza di *Libro* ed una istanza di *Lettore*.

Altre forme di associazioni

- **Associazioni riflessive**

In un Object Diagram possono esistere link tra istanze della stessa classe. Ma c'è una sola classe nel Class Diagram per rappresentare entrambi gli oggetti. In tal caso è usata una associazione riflessiva.



- **Aggregazioni**

Maggior enfasi su come i due oggetti sono correlati nel sistema. Caratterizzata dalla relazione “ha un”. Possono avere molteplicità

- **Composizioni**

Maggior enfasi su come i due oggetti sono correlati nel sistema. Caratterizzata dalla relazione “contiene sempre”. Possono avere molteplicità.

23. Analisi della dinamica del modello

Modellazione Dinamica

Consiste nello specificare come gli oggetti operano e cooperano nel tempo. Essa avviene due volte: durante la fase di analisi iniziale (quando si vuole essere sicuri che le operazioni siano possibili nel sistema) e durante la fase di progetto fisico (per assegnare le descrizioni dei metodi alle classi appropriate). La modellazione dinamica può essere rappresentata in UML tramite i diagrammi: Sequence, Collaboration, State e Activity.

Responsabilità

Nel paradigma OO, "Responsabilità" significa:

- qualcosa che la classe conosce (stato);
- qualcosa che la classe sa (comportamento);
- qualcosa che l'oggetto conosce (stato);
- qualcosa che l'oggetto sa (comportamento).

Un comportamento definito in una interfaccia può essere realizzato in più di una classe. In questo caso si ha una realizzazione polimorfa di una responsabilità.

Il comportamento di una classe può essere distribuito in più sottoclassi oppure realizzato tutto nella stessa classe. La qualità complessiva di progettazione OO è determinata da come e dove vengono distribuite le responsabilità. Questo è un compito cruciale nella programmazione OO, tanto che sono stati sviluppati alcuni principi generali da usare come linee guida per l'assegnazione delle responsabilità (Patterns).

Evoluzione del sistema

Descrivere come cambia nel tempo un sistema significa considerare vari aspetti:

- 1) determinare se ciascuna operazione è una richiesta singola da parte di un utente o di altri sistemi;
- 2) determinare gli oggetti coinvolti in ogni singola operazione, come essi interagiscono e come ciascuna operazione può alterare lo stato interno degli oggetti.

I diagrammi usati in questo caso sono:

- Diagrammi **Sequence** e **Collaboration**
 - Sono usati per descrivere gli scenari di uno stesso Use Case.
 - Durante la fase di Analisi devono riflettere solo le interazioni.
 - Durante la fase di Progettazione, ciascuna interazione sarà convertita nell'invocazione di un metodo nel linguaggio OO scelto.
 - Per un certo Use Case dovrebbero essere prodotti tanti diagrammi Sequence e/o Collaboration, uno per ogni scenario di quello Use Case.
- Diagrammi **State**
 - Mostrano come cambia un oggetto nel tempo a causa dell'invocazione di metodi.
 - Consistono di:
 - Stati: uno stato è l'insieme di valori degli attributi;
 - Eventi: un evento è lo stimolo che in un oggetto causa la transizione da uno stato all'altro.
 - Sono importanti per descrivere gli stati legalmente assumibili dagli oggetti e la natura delle modifiche che possono legittimamente accadere nel tempo.

Java

24. Introduzione a Java

Tecnologia Java

Java è un linguaggio di programmazione, con sintassi simile a C++ e semantica simile a SmallTalk. Java è di solito menzionato (ma non serve solo) per produrre *applet*: applicazioni WEB che risiedono sul server ma sono eseguite direttamente dal browser WEB del cliente.

Le **applicazioni** Java sono programmi autonomi che non richiedono un browser WEB per essere eseguiti, infatti, tali programmi richiedono solo un elaboratore su cui sia installato *Java Runtime Environment* (JRE). Un ambiente di sviluppo, cioè un insieme di numerosi strumenti per la realizzazione di programmi: un compilatore, un interprete, un generatore di documentazione, uno strumento di aggregazione di file, ...

Un esempio

I sorgenti:

- File: *Saluti.java*

```
public class Saluti {  
    private String saluto;  
    Saluti (String s) {  
        saluto = s;  
    }  
    public void saluta (String chi) {  
        System.out.println(saluto + " " + chi);  
    }  
}
```

- File: *ProvaSaluti.java*

```
public class ProvaSaluti {  
    public static void main (String[] args) {  
        Saluti ciao = new Saluti ("Ciao");  
        Saluti arrivederci = new Saluti ("Arrivederci");  
        ciao.saluta("Alberto");  
        ciao.saluta("Marcello");  
        arrivederci.saluta("Stefania");  
    }  
}
```

Compilazione ed esecuzione

- Posto che i due file precedenti siano nella cartella corrente, il comando: \$ javac *ProvaSaluti.java* Produrrà il file oggetto *ProvaSaluti.class*, ma anche il file *Saluti.class*.
- Una esecuzione attraverso l'interprete Java:

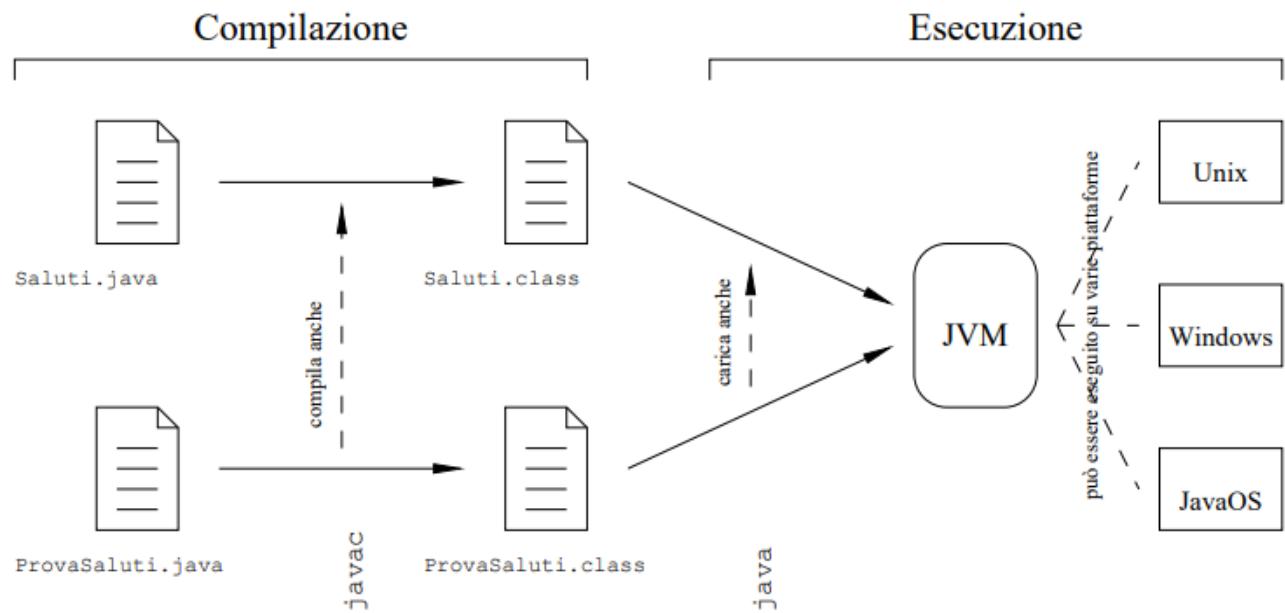
```
$ java ProvaSaluti  
Ciao Alberto  
Ciao Marcello  
Arrivederci Stefania  
$
```

Errori comuni

- Errori in compilazione:
 - Installazione JavaSDK non corretta.
 - Nome di metodi di sistema non corretti.
 - Errata corrispondenza tra nome di classe e nome di file.
 - Numero di classi pubbliche in un file.

- Errori in esecuzione:
 - Errata scrittura della classe da caricare inizialmente in memoria.
 - Errata scrittura della firma del metodo main.

L'esempio esemplificato



25. JVM e JRE

Java Virtual Machine (JVM)

La JVM è una macchina immaginaria che è emulata dalla macchina reale. I programmi per la JVM sono contenuti in file *.class*, ciascuno dei quali contiene al più una classe pubblica.

Quindi, la JVM:

- fornisce le specifiche della piattaforma hardware;
- legge i codici bytecode compilati, che sono indipendenti dalla piattaforma;
- è realizzata in software o in hardware;
- è realizzata come strumento di sviluppo software oppure in un browser Web

Dal punto di vista della realizzazione, la JVM specifica:

- l'insieme di istruzioni della CPU;
- l'insieme dei registri;
- il formato del file Class;
- lo stack di esecuzione;
- lo heap gestito dal garbage collector;
- l'area di memoria.

Dal punto di vista dell'utilizzatore:

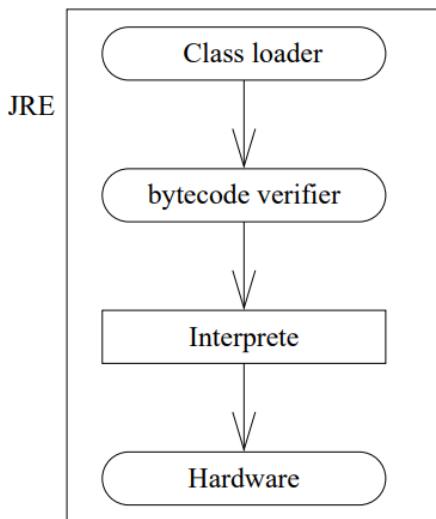
- Il formato del programma compilato, eseguibile dalla JVM, consiste in bytecode molto compatti ed efficienti.
- La maggior parte del type-checking è svolto durante la compilazione.
- Ogni realizzazione della JVM deve essere capace di eseguire un qualunque programma conforme alle specifiche della JVM

Garbage Collector

Il garbage collector è una delle varie misure adottabili per evitare problemi di memory leakage (perdita o fuoriuscita di memoria; aumento dell'occupazione di memoria ingiustificato, dovuto a mancate deallocazioni di strutture dati ormai non più utilizzabili). La memoria allocata che non è più necessaria deve essere resa disponibile; in altri linguaggi la deallocazione è responsabilità del programmatore ma Java fornisce un thread a livello di sistema per tracciare l'allocazione di memoria, ovvero il **Garbage Collector** che ricerca e libera la memoria non più necessaria. Il garbage collector è eseguito automaticamente ed è dipendente dalle varie realizzazioni di JVM.

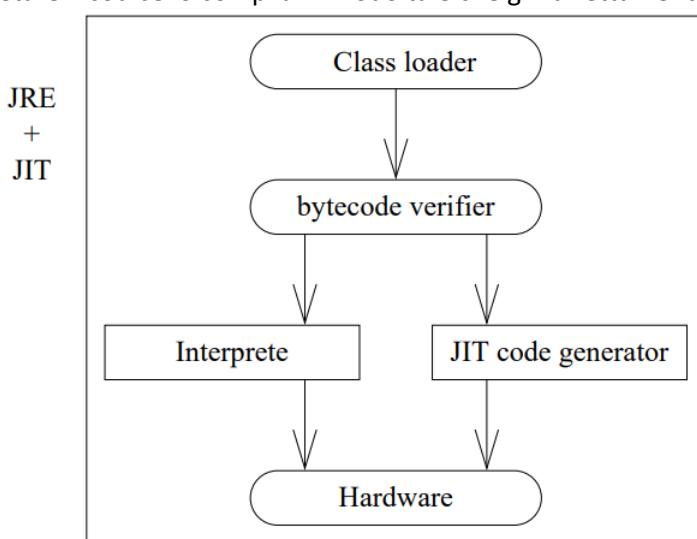
Java Runtime Environment (JRE)

Java Runtime Environment è una realizzazione della Java Virtual Machine. In dettaglio:



Java Runtime Environment e Just in Time Compilation (JRE + JIT)

Per migliorare la velocità di esecuzione la macchina virtuale ha questo JIT code generator che invece di interpretare il codice lo compila in modo tale che giri direttamente sull'hardware:



Sicurezza

Java, ad esempio, limita moltissimo l'utilizzo dei puntatori (a differenza del C++) per evitare errori di allocazione dinamica e soprattutto per evitare che vengano usati per attacchi alla sicurezza. (ad esempio, in C++ potrei forzare un puntatore float a puntare in un area di memoria intera potendo modificare un area di memoria a cui non dovrei avere normalmente accesso).

Miglioramento della sicurezza del codice che proviene da fonti esterne.

- **Il Class Loader:**
 - carica solo le classi necessarie per l'esecuzione del programma;
 - mantiene le classi del file system locale in uno spazio di nomi separato: ciò limita applicazioni "cavallo di Troia", poiché le classi locali sono sempre caricate per prime;
 - previene il così detto spoofing (un tipo di attacco informatico che impiega in varie maniere la falsificazione dell'identità, *spoof*).
 - **Il Bytecode Verifier** controlla che:
 - il codice rispetti le specifiche della JVM;
 - il codice non violi l'integrità del sistema;
 - il codice non generi stack overflow o underflow;
 - non vi siano conversioni di tipo illegali (e.g. la conversione di interi in puntatori).

26. Un po' di sintassi

Classi

La sintassi per la dichiarazione di una classe è:

Esempio:

```
public class Automobile {  
    private double maxPosti;  
    public void setMaxPosti(double valore) {  
        maxPosti = valore;  
    }  
}
```

Attributi

La sintassi per la dichiarazione di un attributo è:

```
< attribute_declarator > ::=  
    < modifier > < type > <name> [= < default_value >];  
  
<type > ::= byte | short | int | long | char |  
           float | double | boolean | <class >
```

Esempio:

```
public class Foo {  
    public int x;  
    public float y = 10000.0 F;  
    private String nome = "Marcello Sette";  
}
```

Metodi

La sintassi per la dichiarazione di metodi è:

```
< method_declarator > ::=  
  < modifier > < return_type > <name> (< parameter >*) {  
    ...  
    < statement >*  
  }
```

Esempio:

```
public class Cosa {  
    private int x;  
    public int getX() {  
        return x;  
    }  
    public void setX(int nuovo_x) {  
        x = nuovo_x;  
    }  
}
```

Accesso a membri oggetto

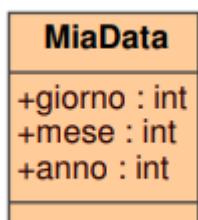
La notazione dot: `< object >. < member >` è usata per l'accesso a membri **non privati** di un'oggetto.

Esempio:

```
public class TestCosa {  
    public static void main (String[] args) {  
        Cosa cc = new Cosa();  
  
        cc.setX(47);  
        System.out.println (" cc.x vale " + cc.getX());  
    }  
}
```

27. Incapsulazione

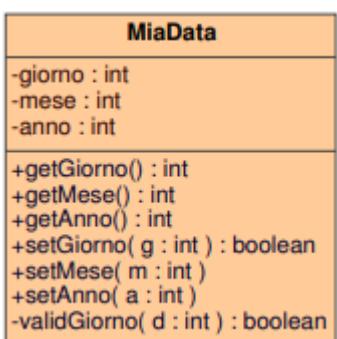
Il problema



Il codice cliente ha accesso diretto ai dati interni:

```
MiaData d = new MiaData();  
d.giorno = 32;  
// non valido  
d.mese = 2; d.giorno = 30;  
// plausibile ma sbagliato  
d.giorno = d.giorno + 1;  
// nessun controllo del limite superiore
```

La soluzione



Il cliente deve usare i metodi per accedere ai dati interni:

```
MiaData d = new MiaData ();  
d.setGiorno (32);  
// non valido , ma ritorna falso  
d.setMese (2);  
d.setGiorno (30);  
// plausibile ma sbagliato ; setGiorno ritorna false  
d.setGiorno (d.getGiorno () + 1);  
// restituisce false se si eccede il limite
```

28. Costruttori

Sintassi

La sintassi per la dichiarazione di un costruttore è:

```
< constructor_declaraction > ::=  
< modifier > < class_name > (< parameter >*) {  
    < statement >*  
}
```

Esempio:

```
public class Cosa {  
    private int x;  
    public Cosa() {  
        x = 47;  
    }  
    public Cosa (int nuova_x) {  
        x = nuova_x;  
    }  
}
```

Attenzione:

- i costruttori NON sono metodi;
- i costruttori NON sono ereditati;
- un costruttore NON ha valore di ritorno;
- gli unici modificatori validi per il costruttore sono:
public, protected e private

Costruttore di default

- C'è sempre almeno un costruttore in ogni classe.
- Se il programmatore non scrive nessun costruttore, allora verrà usato un costruttore di default.
- Il costruttore di default non ha argomenti.
- Il costruttore di default non ha corpo.
- Il costruttore di default permette di creare istanze di classi (con l'espressione *new Xxx()*) senza dover scrivere un costruttore.

Attenzione: Se si aggiunge una dichiarazione di costruttore con argomenti ad una classe che in precedenza non aveva costruttori esplicativi, allora si perde il costruttore di default. Da quel punto in poi, una chiamata a *new Xxx()* genererà un errore di compilazione.

29. File sorgenti, pacchetti e cartelle

Sorgenti

```
< source_file > ::=  
[< package_declaraction >]  
< import_declaraction >*  
< class_declaraction >*
```

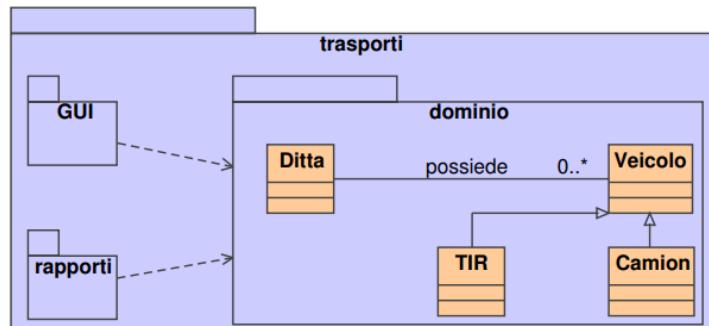
- l'ordine degli elementi è importante;
- il nome del file sorgente deve essere lo stesso dell'unica classe pubblica contenuta nel file; se il file sorgente non contiene classi pubbliche allora il nome del file può essere qualunque.

Esempio: il file *RapportoCapacitaVeicolo.java*

```
package trasporti.rapporti.Web ;  
  
import trasporti.dominio.*;  
import java.util.List;  
import java.io.*;  
  
public class RapportoCapacitaVeicolo {  
    private List veicoli;  
    public void generaRapporto(Writer output) {  
        ...  
    }  
}
```

Pacchetti

Possono contenere classi e sotto-pacchetti, i pacchetti aiutano a gestire grandi sistemi software.



- **Enunciato package**

La sintassi dell'enunciato package è:

```
< package_declaration > ::= package < top_pkg_name > [.< sub_pkg_name >]*;
```

ove:

- la dichiarazione specifica che il contenuto del file appartiene al pacchetto dichiarato;
- è permessa una sola dichiarazione di pacchetto per file sorgente;
- si deve specificare la dichiarazione all'inizio del file sorgente;
- se non è dichiarato il pacchetto, allora il contenuto del file appartiene al pacchetto di default;
- i nomi di pacchetto devono essere gerarchici e separati da punti;
- un nome di pacchetto corrisponde di solito ad un nome di cartella;
- in genere i nomi di pacchetto sono scritti in lettera minuscola (my_pack), mentre i nomi di classe iniziano per lettera maiuscola (My_class).

- **Enunciato import**

- La sintassi dell'enunciato import è:

```
< import_declaration > ::= import < top_pkg_name > [.< sub_pkg_name >]*.< classes >;
```

```
< classes > ::= < class_name > | *
```

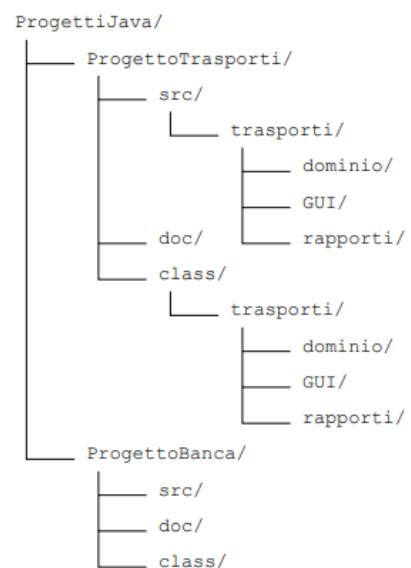
- Precede ogni dichiarazione di classe.
- Dice al compilatore dove trovare le classi da usare.
- Importa solo lo spazio dei nomi, non le classi. Serve cioè solo ad abbreviare la scrittura del codice, permettendo di citare solo il nome di una classe e non obbligatoriamente tutto il percorso per raggiungerla.

Esempio:

```
import trasporti.dominio.*;
import java.util.List ;
import java.io.*;
```

Cartelle e pacchetti

Un organizzazione consigliata è quella di avere i pacchetti contenuti in alberi di cartelle, dove i nomi di cartelle sono uguali ai nomi di pacchetto. È buona norma di programmazione quella di separare, nello sviluppo di un progetto, i file sorgente dai file compilati. Un metodo possibile è il seguente:



Come?

- Normalmente il compilatore pone i file .class nella stessa cartella dei file .java
- I file .class possono essere re-indirizzati in un'altra cartella usando l'opzione `-d` del comando javac.
- Se si sta compilando un insieme di file all'interno della gerarchia dei pacchetti (cioè non nella cartella principale dei sorgenti), allora bisogna anche specificare l'opzione `-sourcepath`

Per esempio:

```
$ cd ProgettiJava / ProgettoTrasporti / src / trasporti / dominio
$ javac - sourcepath ProgettiJava / ProgettoTrasporti / src \
      ... -d ProgettiJava / ProgettoTrasporti / class *. java
```

Fase di rilascio

Una applicazione può essere rilasciata su una macchina cliente in più modi:

- 1) Se si utilizza un file JAR, quel file deve essere copiato nella cartella delle "estensioni di libreria". Per esempio:

Ambiente Linux: /usr/jdk1.4/jre/lib/ext/

Ambiente Windows: C:\jdk1.4\jre\lib\ext\

- 2) Se l'applicazione è rilasciata come una gerarchia di file .class, allora bisogna porre tutta la gerarchia all'interno della cartella delle "classi JRE". Per esempio:

Ambiente Linux: /usr/jdk1.4/jre/classes/

Ambiente Windows: C:\jdk1.4\jre\classes\

Uso della documentazione: Dimostrazione sull'utilizzo. Descrizione della gerarchia delle classi della API.

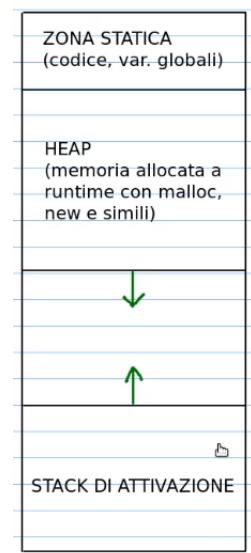
30. Ambiente non locale dal paradigma imperativo a quello a oggetti

Organizzazione della memoria nei linguaggi di programmazione

La memoria viene organizzata in una zona statica dove è situato il codice e le variabili globali; una zona di memoria che cambia dinamicamente conosciuta come heap, dove la memoria viene allocata a runtime con malloc, new e simili; ed infine la zona di memoria dello stack di attivazione.

Quel che si fa è organizzare la memoria iniziando ad allocare la zona statica seguita dallo heap, quest'ultimo viene fatto eventualmente crescere di dimensione, come succede con le chiamate ricorsive dello stack di attivazione. Inoltre, lo heap e lo stack crescono in due direzioni opposte così da ottimizzare di molto la memoria; nel caso i due stack si incontrassero ci sarà un errore di out of memory.

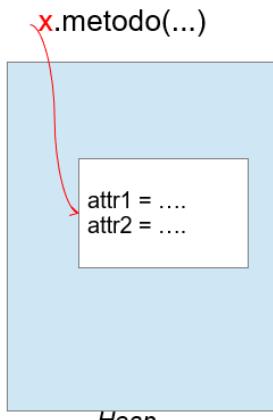
Nel paradigma ad oggetti però le cose cambiano, infatti gli oggetti stessi vengono allocati nello heap, più precisamente le variabili degli oggetti sono allocate nella heap invece che nello stack (come succede nel paradigma imperativo).



Ambiente non locale nel paradigma imperativo

L'ambiente non locale è interamente contenuto nello stack di attivazione ed è interamente determinato dallo scoping (statico o dinamico).

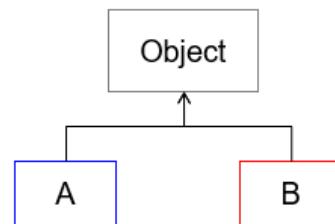
Ambiente non locale nel paradigma a oggetti



Consiste negli attributi dell'oggetto che esegue il metodo ed è ancora una forma di scoping statico (attributi dichiarati nel blocco-classe che contiene il metodo) ma è allocato nello heap.

In questo caso l'ambiente non locale viene indicato esplicitamente con il destinatario del messaggio, che se non viene specificato è rappresentato dall'argomento *this*.

```
class A {  
    type1 attr1;  
    class B {  
        type2 attr2;  
        metodo(...)  
    }  
}
```



31. Identifieri, parole chiavi e tipi primitivi

Commenti

Tre stili di scrittura di commento al codice:

```
// commento su una singola riga  
  
/* commento su una  
   o più righe */  
  
/** commento usato per la documentazione automatica  
   del codice tramite l'uso dello strumento javacod */
```

Blocchi

Un *enunciato* è costituito da una o più righe di codice terminate da un ';' mentre un *blocco* è la collezione di enunciati racchiusi tra parentesi graffe:

- I blocchi possono essere annidati uno nell'altro.
- Il numero di spazi o righe bianche è ininfluente.

```
{  
    x = y + 1;  
    y = x + 1;  
}
```

Identifieri

Gli identifieri sono nomi assegnati a variabili, classi o metodi, possono cominciare con un carattere Unicode, underscore (_), oppure dollaro (\$); sono **case sensitive** e non hanno una lunghezza massima.

Fanno eccezione i nomi di una classe che possono essere costituiti solo da caratteri ASCII per evitare problemi di portabilità poiché non molti sistemi supportano Unicode nei nomi dei file (il nome della classe deve corrispondere a quello del file).

Parole chiavi

abstract	continue	float	long	short	transient
boolean	default	for	native	static	true
break	do	goto	new	strictfp	try
byte	double	if	null	super	void
case	else	implements	package	switch	volatile
catch	extends	import	private	synchronized	while
char	false	instanceof	protected	this	
class	final	int	public	throw	
const	finally	interface	return	throws	

- Non possono essere usate come identifieri.
- *true*, *false*, *null* sono in minuscolo, non in maiuscolo come in C++. Strettamente parlando sono litterali, non parole chiavi.
- Non c'è l'operatore *sizeof*: la dimensione e la rappresentazione dei tipi è fissa e non dipende dalla realizzazione della JVM.
- *goto* e *const* sono parole chiavi che non sono usate in Java ma non è possibile definirle come nomi.

Tipi primitivi

In Java esistono otto tipi primitivi:

- Logici: *boolean*
 - Il tipo boolean ha due litterali: *true*, *false*. Non c'è cast tra tipi interi e boolean, infatti interpretare valori numerici come valori logici non è permesso in Java.
- Testuali: *char*
 - Rappresenta un carattere Unicode (16 bit). I litterali di questo tipo sono inclusi tra apici singoli come ad esempio la lettera greca phi '\u03A6'.

- Le stringhe non sono tipi primitivi
 - String è una classe (comincia per lettera maiuscola) ed "Ha i suoi litterali racchiusi tra apici doppi". Esempio *String saluto = "Buon giorno !! \n ";*
- Interi: *byte, short, int, long*
 - Rappresentano rispettivamente i valori:

Lunghezza	Tipo	Range
8 bit	byte	$-2^7 \dots 2^7 - 1$
16 bit	short	$-2^{15} \dots 2^{15} - 1$
32 bit	int	$-2^{31} \dots 2^{31} - 1$
64 bit	long	$-2^{63} \dots 2^{63} - 1$

I litterali hanno tre forme: decimale, ottale ed esadecimale.

- 2 il valore decimale è due.
- 077 lo zero iniziale denota un valore ottale.
- 0xBAAC la parte 0x iniziale denota un valore esadecimale.

Assumono come tipo di default int.

Suffisso *L* oppure *l* se si vuole che siano di tipo long ad esempio *2L*

Quando si assegna il valore di un litterale ad una variabile, il compilatore determina la dimensione del litterale a seconda della variabile mentre quando si assegna il valore di una espressione ad una variabile, la dimensione del valore non è modificabile poiché il compilatore non può conoscere il valore dell'espressione a run time e quindi non sa se è nel range ad esempio dello short (si può fare esplicitando il cast).

- Floating point: *float, double*

- Rappresentano i valori:

Lunghezza	Tipo
32 bit	float
64 bit	double

Suffisso *F* o *f* per litterali del tipo float.

Suffisso *D* o *d* per litterali del tipo double anche se è ridondante visto che i litterali a virgola mobile hanno come tipo di default il double, quindi se non specificato il tipo è double.

32. Tipi reference

Cosa sono

Tutti i tipi non primitivi sono tipi reference ed una variabile di questo tipo contiene la "maniglia" (*handle*) di un oggetto; la scelta di handle invece che pointer in java è per rimarcare la differenza che ha con gli altri

```
public class MiaData {
    private int giorno = 1;
    private int mese = 1;
    private int anno = 2006;
}

// La classe MiaData può essere usata in questo modo:
public class TestMiaData {
    public static void main ( String [] argv ) {
        MiaData oggi = new MiaData();
    }
}
```

linguaggi, infatti java non permette l'aritmetica tra puntatori ma l'unica operazione consentita è la new, dove java controlla se il puntatore corrisponde al tipo corretto, oltre alla costruzione ed al passaggio non è consentito altro (ad esempio non è possibile la conversione di tipo).

Costruzione di oggetti

`new Xxx()` serve ad allocare spazio per il nuovo oggetto. Scatena i seguenti processi:

- 1) Viene allocato lo spazio per il nuovo oggetto e le variabili dell'istanza sono inizializzate al loro valore di default (e.g. 0, `false`, `null`, e così via).
- 2) Viene eseguita ogni inizializzazione esplicita degli attributi.
- 3) Viene eseguito un costruttore.
- 4) Viene assegnato il riferimento finale all'oggetto.

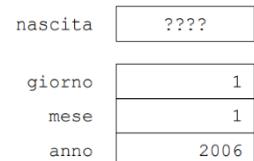
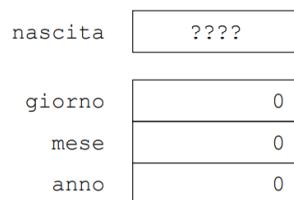
Prima di esaminare separatamente ciascuna di queste fasi, mostrando ciò che succede quando viene seguito il codice: `MiaData nascita = new MiaData(23, 4, 1964);`; si fa presente che il processo di costruzione di oggetti e della loro inizializzazione è notevolmente più complesso di come verrà descritto:

- **Allocazione di memoria**

La dichiarazione `MiaData nascita` causa l'allocazione dello spazio memoria del solo riferimento (non ancora inizializzato):

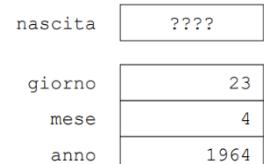


L'uso della parola chiave `new`, alloca spazio per `MiaData`:



- **Inizializzazione degli attributi**

Successivamente, per gli attributi sono usate le inizializzazioni esplicite all'interno della classe, quelle che eventualmente sono scritte nel momento della definizione dell'attributo (inizializzazione dell'oggetto da parte del progettista della classe):

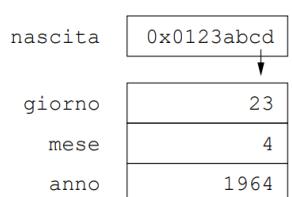


- **Esecuzione del costruttore**

Ora è invocato il costruttore. Con esso si possono sostituire inizializzazioni personali dell'utente dell'oggetto a quelle di default previste dal progettista della classe. Si possono anche passare argomenti, così che il codice che richiede le costruzione del nuovo oggetto possa controllare l'oggetto che verrà creato:

- **Assegnazione del riferimento**

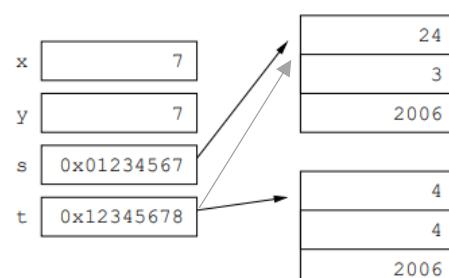
L'assegnazione, infine, inserisce l'indirizzo del nuovo oggetto nella locazione del riferimento:



Esempio

```
int x = 7;
int y = x;

MiaData s = new MiaData(24, 3, 2006);
MiaData t = s //freccia grigia
t = new MiaData(4, 4, 2006);
```



33. Parametri

Passaggio per valore

Java permette il passaggio dei parametri per valore (nella nostra tassonomia, parametri IN realizzati per copia). Il passaggio per riferimento (che permette la modifica del valore del parametro nel contesto della procedura chiamante) è **proibito** in java, infatti quando si passa una istanza di oggetto come argomento di un metodo, quello che si sta passando non è l'oggetto, ma solo un riferimento a quell'oggetto.

Questo riferimento è copiato nel parametro formale. Attenzione che, in quest'ultimo caso, sarà possibile la modifica nel contesto del chiamante (mai del valore della variabile) dell'oggetto a cui fa riferimento quella variabile.

Il riferimento *this*

La parola chiave this può essere usata:

- Per fare riferimento, all'interno di un metodo o di un costruttore locale, ad attributi o metodi locali. Questa tecnica è usata per risolvere ambiguità in alcuni casi in cui una variabile locale di un metodo maschera un attributo locale dell'oggetto.
- Per permettere ad un oggetto di passare il riferimento a sé stesso come parametro ad un altro metodo o costruttore.

34. Variabili ed espressioni

Ambiti

Le variabili definite all'interno di un metodo sono dette **locali**; esse sono create quando il metodo è invocato e sono distrutte alla sua terminazione; esse devono essere inizializzate esplicitamente prima di essere usate. Le variabili usate come parametro di metodi, visto il meccanismo di passaggio di parametri (IN per copia), sono variabili locali.

Le variabili definite all'esterno di un metodo sono create quando viene costruito un oggetto. Ve ne sono di due tipi:

- **Variabili di classe:** sono dichiarate usando il modificatore *static*; sono create nel momento in cui una classe è caricata in memoria (esistono a prescindere dalle istanze); continuano ad esistere finché la classe rimane in memoria; una variabile di classe è valida per tutte le istanze.
- **Variabili di istanza:** quelle dichiarate senza modificatore *static*; sono create durante la costruzione di una istanza e continuano ad esistere finché l'oggetto esiste; di conseguenza abbiamo una variabile diversa per ciascuna istanza.

Inizializzazioni

A priori possiamo dire che nessuna variabile può essere usata prima di essere inizializzata; detto ciò, si fa distinzione tra le variabili non locali e le variabili locali:

- Le variabili non locali sono inizializzate automaticamente dalla JVM, nel momento in cui la classe è caricata in memoria o in cui è allocato spazio per il nuovo oggetto, ai seguenti valori:

Variabile	Valore
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000'
boolean	false
riferimenti	null

- Le variabili locali (quelle dei metodi) devono essere inizializzate manualmente prima dell'uso.

Espressioni

Gli operatori sono simili a quelli di C o C++ quindi ci soffermeremo sulle differenze piuttosto che sul costrutto in sé (a destra una tabella di tutti gli operatori in ordine crescente di precedenza):

Operatori logici:

- Operatori booleani: ! (NOT), & (AND), | (OR), ^ (XOR)
- Operatori booleani con corto circuito: || (OR), && (AND). Calcolano come l'OR e l'AND ma la valutazione si ferma prima di dare la risposta, quindi ad esempio se il primo argomento di || risulta vero il secondo non viene calcolato. È utile nel caso non sia possibile giudicare la seconda condizione a priori, ad esempio: (d != null) && (d.giorno > 31)

Associat.	Operatori
R → L	++ -- + - ~ !
L → R	* / %
L → R	+ -
L → R	<< >> >>>
L → R	< > <= >= instanceof
L → R	== !=
L → R	&
L → R	^
L → R	
L → R	&&
L → R	
R → L	? :
R → L	= *= /= %= += -=
	&= ^= = <=>= >>=

Operatori di bit:

- Operazioni di manipolazioni di bit su interi: ~ (Complemento ad 1), & (AND), | (OR), ^ (XOR), in questo caso il risultato sarà un intero in cui i bit vengono modificati nel modo seguente (come se lo 0 fosse true e l'1 false):

```
byte a = 45;
byte b = 79;
System.out.println(~b = " + (byte) (~b)); // -80
System.out.println("a & b = " + (byte) (a & b)); // 13
System.out.println("a ^ b = " + (byte) (a ^ b)); // 98
System.out.println("a | b = " + (byte) (a | b)); // 111
```

$\sim \begin{array}{ c c c c c c c c } \hline 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array}$	$\& \begin{array}{ c c c c c c c c } \hline 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ \hline \end{array}$	$\wedge \begin{array}{ c c c c c c c c } \hline 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array}$	$\vee \begin{array}{ c c c c c c c c } \hline 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ \hline \end{array}$
$\begin{array}{ c c c c c c c c } \hline 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ \hline \end{array}$	$\begin{array}{ c c c c c c c c } \hline 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array}$	$\begin{array}{ c c c c c c c c } \hline 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ \hline \end{array}$	$\begin{array}{ c c c c c c c c } \hline 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ \hline \end{array}$

Operatori right shift:

- Nello shift a destra aritmetico o con segno (>>) il bit di segno più significativo viene conservato:
 - 128 >> 1 vale 64

$$\begin{array}{r} 00000000 00000000 00000000 10000000 = 128 \\ 00000000 00000000 00000000 01000000 = 64 \\ \hline 00000000 10000000 = 128 \\ 11111111 01111111 + 1 \\ 11111111 10000000 = -128 \end{array}$$
 - $-128 >> 3$ vale -16

$$11111111 11110000 = -16$$

(il negativo si forma con il complementare + 1; mentre per lo shift traslano a destra tutti i valori successivi alla cifra significativa di 3 posizioni)
- Nello shift a destra logico o senza segno (>>>) il bit di segno è azzerato nello spostamento:
 - $-1 >>> 30$ vale 3

$$\begin{array}{r} 11111111 11111111 11111111 11111111 = -1 \\ 00000000 00000000 00000000 00000011 = 3 \end{array}$$
- Se l'operando di sinistra è un int, prima di applicare lo shift viene ridotto l'operando di destra modulo 32; se l'operando di sinistra è un long, prima di applicare lo shift viene ridotto l'operando di destra modulo 64. Infatti, supponendo di fare $x >>> 64$, il valore di x non viene modificato.
- L'operatore >>> è ammesso solo per i tipi interi *int* e *long*; se viene usato per *short* o *byte*, questi valori sono promossi, con estensione di segno, ad *int* prima di applicare lo shift.

Operatori left shift:

- Per lo shift a sinistra non c'è il problema del bit di segno:

$11111111\ 11111111\ 11111111\ 11111111 = -1$
 $11111111\ 11111111\ 11111111\ 11111110 = -2$

- $-1 \ll 1$ vale -2 :

Concatenazione di stringhe:

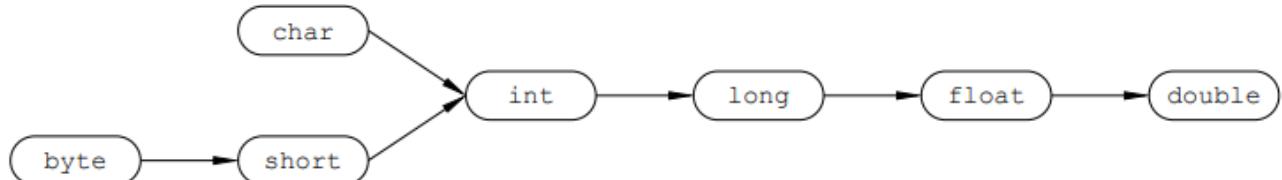
- L'operatore `+` esegue la concatenazione di due oggetti `String`, producendo un nuovo oggetto `String`; ad esempio `"Valentino" + " " + "Rossi"` avrà come output: `Valentino Rossi`
- Un argomento deve essere un oggetto `String`.
- Gli altri argomenti sono convertiti a `String` automaticamente, invocando il metodo `toString`.

35. Casting di primitivi

Conversioni implicite

Una *conversione di tipo per assegnazione* avviene quando si assegna un valore ad una variabile di tipo differente e ciò può avvenire o con un operatore di assegnazione, o durante il passaggio di parametri ad un metodo (il tipo `boolean` non può essere mai convertito).

Le conversioni legali (ovvero quelle che avvengono automaticamente) sono quelle per cui esiste un percorso nel seguente grafo:



Conversioni esplicite

Regola: se è possibile la perdita di informazioni in una assegnazione o in un passaggio di parametri, il programmatore dove confermare l'assegnazione con un “cast” esplicito. In un cast esplicito il valore in eccesso viene troncato dalla cifra più significativa, dunque bisogna assicurarsi che non ci siano perdite di informazioni. Ad esempio, l'assegnazione da `long` a `int` richiede un cast esplicito:

```
long grande = 77L;
int piccolo = grande; // Sbagliato, richiede cast
int piccolo = ( int ) grande; // OK
int piccolo = 77L; // Sbagliato, richiede cast
int piccolo = ( int ) 77L; // OK, ma ...
int piccolo = 77; // pure OK
// ( default : 77 letterale int )
```

Eccezione: : Il casting esplicito non è necessario per i litterali interi (non i floating point) che ricadono nel range legale del tipo di destinazione:

```
int i = 12;
byte b = 12; // OK
byte b = 128; // Illegale : è superiore a 127
byte b = i; // Illegale : vale 12 ma non è un litterale
float x = 3.14; // Illegale : eccezione non valida per float point
```

Promozione aritmetica

In una espressione aritmetica, le variabili sono automaticamente promosse ad una forma più estesa per non causare perdita di informazioni.

- Le regole per operatori unari:
 - Se l'operando è un *byte*, uno *short* o un *char*, esso è convertito ad *int* (a meno che l'operatore non sia *+* o *-*, nel qual caso non avviene nessuna conversione).
 - Se l'operatore non è più piccolo di *int* non avviene nessuna conversione.
- Le regole per operatori binari:
 - Se uno degli operandi è un *double*, allora l'altro operando è convertito a *double*.
 - Se uno degli operandi è un *float*, allora l'altro operando è convertito a *float*.
 - Se uno degli operandi è un *long*, allora l'altro operando è convertito a *long*.
 - Altrimenti, entrambi gli operandi sono convertiti ad *int*.

36. Costrutti e controlli

Enunciati branch

if, else

```
if (<boolean expression>) {
    <statement>*
}

if (<boolean expression>) {
    <statement>*
} else if (<boolean expression>) {
    <statement>*
} else {
    <statement>*
}
```

Attenzione all'espressione condizionale: deve essere una espressione booleana, non un intero come in c/c++.

switch

```
switch (<expr>) {
    case <constant1>:
        <statement>*
        break;
    case <constant2>:
        <statement>*
        break;
    default:
        <statement>*
        break;
}
```

Dove:

- *<expr>* deve essere compatibile per assegnazione con *int* (*byte*, *short*, *char* sono promossi automaticamente);
- l'etichetta opzionale *(default)* è usata per specificare il segmento di codice da eseguire quando il valore di *<expr>* non corrisponde a nessuno dei valori delle stanze *case*;
- se non è presente l'enunciato opzionale *break*, l'esecuzione continua nella stanza *case* successiva.

Enunciati loop

for

```
for (<init_expr>; <bool_expr>; <alt_expr>) {
    <statement>*
}
```

while

```
while (<bool_expr>) {
    <statement>*
}
```

do/while

```
do {
    <statement>*
} while (<bool_expr>);
```

Controlli speciali

break [label]

usata per uscire in modo prematuro da un enunciato switch, da enunciati di loop o da blocchi etichettati.

continue [label]

usata per saltare direttamente alla fine del corpo di un loop.

label : (statement)

usata per identificare un qualunque enunciato verso il quale può essere trasferito il controllo.

Questionario (soluzioni in fondo)

D1: In questa successione di enunciati, quale linea non compila?

- | | |
|---------------------|-------------------|
| A. byte b = 5; | F. b = s; |
| B. char c = '5'; | G. i = c; |
| C. short s = 55; | H. if (f > b) ... |
| D. int i = 555; | I. f = i |
| E. float f = 555.5f | |

D2: Il codice seguente viene compilato correttamente?

```
byte b = 2;
byte bl = 3;
b = b * bl ;
```

D3: Nel codice seguente, quali sono i possibili tipi per la variabile *result*?

```
byte b = 11;
short s = 13;
result = b * ++s;
```

A. byte, short, int, long, float, double
B. boolean, byte, short, char, int, long, float, double
C. byte, short, char, int, long, float, double
D. byte, short, char
E. int, long, float, double

D3: Dato il codice seguente, quale delle seguenti affermazioni è vera?

```
1 class Cruncher {
2     void crunch(int i) {
3         System.out.println("int version");
4     }
5     void crunch (String s) {
6         System.out.println("String version");
7     }
8
9     public static void main (String args[]) {
10        Cruncher crun = new Cruncher();
11        char ch = 'p';
12        crun.crunch(ch);
13    }
14 }
```

- A. La linea 5 non compila, poiché i metodi void non possono essere sovrapposti.
- B. La linea 12 non compila, poiché nessuna versione di *crunch()* ha un argomento char.
- C. Il codice compila correttamente, ma viene lanciata una eccezione alla linea 12.
- D. Il codice compila correttamente e viene prodotto il seguente output: *int version*
- E. Il codice compila correttamente e viene prodotto il seguente output: *String version*

D1: F perché potrei perdere informazioni nell'assegnare ad un byte uno short. **D2:** non compila poiché il risultato della moltiplicazione restituisce un intero (visto che durante le espressioni vengono tramutati in interi), dovrei fare un cast esplicito. **D3:** E poiché nella moltiplicazione entrambi vengono promossi interi, il risultato dell'espressione è quindi intero, dunque affinché *result* sia possibile senza cast espliciti deve essere di tipo int o superiore. **D4:** D essendo possibile la conversione di char ad intero viene applicato il primo metodo.

37. Array

Dichiarazione, creazione ed inizializzazione

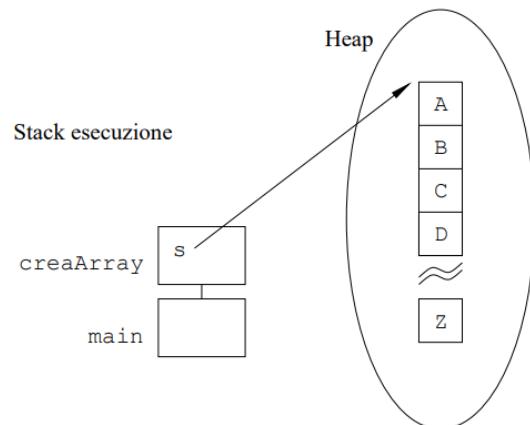
Si possono dichiarare array di tipi primitivi o di riferimenti ad oggetti: `char s[]`; `Point p[]`; oppure `char [] s`; e `Point [] p`. Un array è un oggetto: le dichiarazioni precedenti infatti creano solo il riferimento al rispettivo oggetto. Nella dichiarazione non si deve specificare la dimensione dell'array e se si dichiarano più array in uno stesso enunciato usando le parentesi quadre a sinistra, le parentesi sono applicate a tutte le variabili alla loro destra. Esempio:

```
int a[], b;      // a è riferimento ad array , b è int
int [] a, b;    // a e b entrambi riferimenti ad array
```

L'oggetto array non viene generato nella fase della dichiarazione, infatti per creare l'array si deve usare `new`:

```
public char [] creaArray() {
    char [] s;
    s = new char [26];
    for (int i = 0; i < 26; i++) {
        s[i] = (char) ('A' + i);
    }
    return s;
}
```

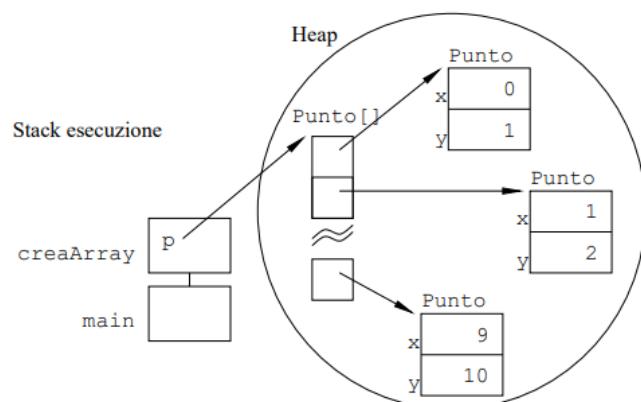
`s = new char[26];` crea un array di 26 caratteri: essi sono inizializzati al loro valore di default ('\u0000' per char). I valori sono accessibili nel range da 0 a 25: ogni tentativo di accesso oltre questo range causerà il lancio di una eccezione a runtime.



Altro esempio per un array di oggetti:

```
public Punto [] creaArray() {
    Punto[] p;
    p = new Punto [10];
    for (int i = 0; i < 10; i++) {
        p[i] = new Punto(i, i + 1);
    }
    return p;
}
```

`p = new Punto[10];` anche questa volta crea un array di 10 riferimenti ad oggetti Punto: essi sono inizializzati al loro valore di default null. Questa volta è cruciale la creazione dei singoli oggetti, prima del loro uso.



Nota: una sintassi alternativa per il metodo precedente poteva essere: `public Punto creaArray() []`

Poiché l'inizializzazione delle variabili è cruciale, Java fornisce due metodi abbreviati per gli array (oltre il metodo diretto che utilizza un loop). Il primo consiste nella dichiarazione, costruzione ed inizializzazione in una riga: che è equivalente a:

```
String [] nomi = {
    "Antonio",
    "Marcello",
    "Anna"
};
```

```
String [] nomi;
nomi = new String[3];
nomi[0] = "Antonio";
nomi[1] = "Marcello";
nomi[2] = "Anna";
```

Il secondo metodo è la costruzione ed inizializzazione di un array anonimo. Esempio:

```
int [] esempiol;
esempiol = new int [] {4, 7, 3};
```

Ma, ancora meglio:

```
public class A {  
    void prendiArray(int [] unArray) {  
        // usa unArray  
    }  
    public static void main(String [] args) {  
        A a = new A();  
        a.prendiArray (new int [] {3 ,4 ,5 ,6 ,7});  
    }  
}
```

Attenzione: non si deve specificare la dimensione di una creazione di un array anonimo, ad esempio la seguente linea di codice è illegale: `new Object [2] {null,new Object()}`;

Esempio

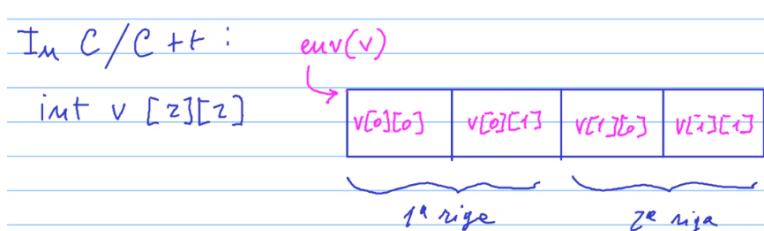
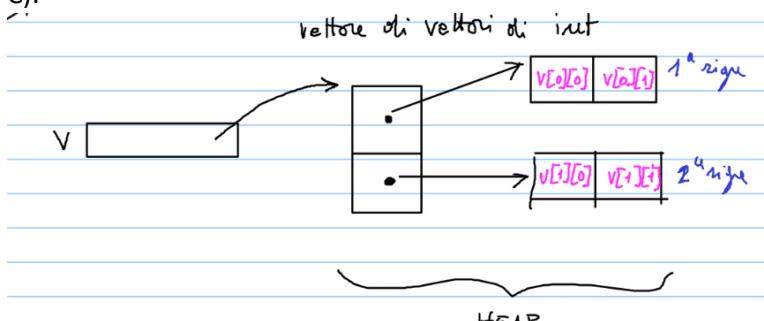
In Java tutti i parametri dei metodi sono di tipo IN (realizzati) per copia. Questo significa che il valore del parametro nel metodo chiamante non può essere modificato. Tuttavia, è possibile creare un riferimento a tale parametro e realizzare ugualmente la modifica nel metodo chiamante. Per esempio, per creare un riferimento ad un tipo primitivo:

```
public class PrimitiveReference {  
    public static void main(String args[]) {  
        int [] mioValore = {1};  
        modifica(mioValore);  
        System.out.println("mioValore contiene " + mioValore [0]);  
    }  
    public static void modifica ( int [] valore ) {  
        valore [0]++;
    }  
}
```

In questa classe abbiamo un metodo che incrementa il primo elemento di un array di interi: nel main creo un array di un elemento che poi viene modificato dal metodo che lo modificherà poiché ho passato l'array per riferimento e quindi mi stamperà in output il valore 2.

Multidimensioni

Con array multidimensionali si intendono semplicemente array di array, per cui se scrivo ad esempio `int v[][] = new int[2][2];` crea una matrice 2x2, all'interno avvengono i seguenti processi (differenti dal C):



Un esempio particolare di costruzione di un array di array; posso creare un oggetto definendo ad esempio solo il numero di righe per poi costruirle in seguito:

```
int dueDim [][] = new int [3][];
dueDim [0] = new int [5];
dueDim [1] = new int [5];
dueDim [2] = new int [5];

int dueDim [][] = new int [][][3]; // illegale
```

Posso benissimo creare una matrice sparsa (che non ha lo stesso numero di elementi a riga), infatti:

```
int dueDim [][] = new int [3][];
dueDim [0] = new int [2];
dueDim [1] = new int [5];
dueDim [2] = new int [8];
```

Volendo sia costruire che inizializzare l'array il procedimento è il seguente:

```
int dueDim[][] = new int[4][5]; // Array rettangolare di array

int [][] multiID = {{5, 4, 3, 2}, {9, 8}, {7, 6, 5}}; // Costruzione e inizializzazione
```

Estremi ed assegnazioni

Indice iniziale è 0 mentre il numero di elementi è parte dell'oggetto array, nell'attributo *Length* (attributo dell'array non modificabile, si può solamente leggere). Esempio:

```
int m[][] = new int[10][5];
System.out.println("m.length vale " + m.length);           // stampa 10
System.out.println("m[0].length vale " + m[0].length ); // stampa 5
```

Essendo l'array un oggetto immutabile, un accesso oltre i limiti causa il lancio di una eccezione a runtime.

In un assegnazione l'array deve avere lo stesso numero di dimensioni della variabile di riferimento per essere corretta (almeno nella maggior parte dei casi); per esempio:

```
int [] vettore;
int [][] matrice = new int [3][];

vettore = matrice; // illegale

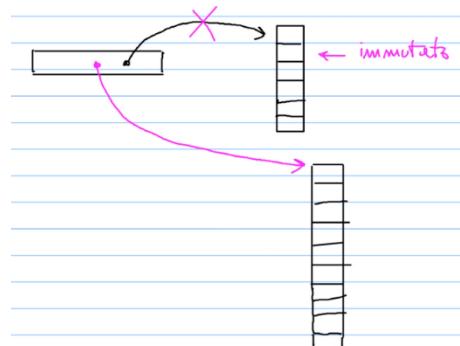
int [] v = new int [6];
vettore = v; // OK
```

Ridimensionamento

Gli array non sono ridimensionabili ma è possibile usare la stessa variabile per riferirsi ad un nuovo array, come ad esempio:

```
int elements[] = new int [6];
elements = new int [10];
```

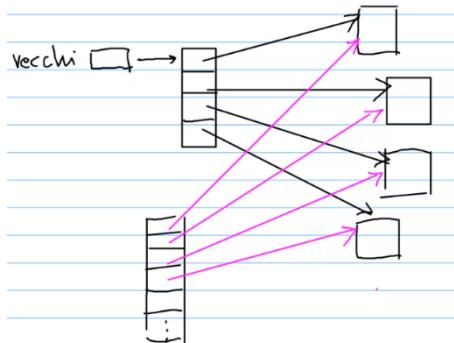
Dove, in questo caso, la variabile *elements*, inizializzata ad un vettore di 6 interi nella prima istruzione, viene costruita con un nuovo vettore di 10 elementi ed inizializzo *elements* ad un nuovo puntatore (non modifico il vecchio oggetto), di conseguenza se il vecchio vettore non ha un'altra variabile che punta ad esso, diverrà irraggiungibile e quindi riciclato dal garbage collector.



Copia

Il metodo **System.arraycopy()**:

```
// array originale  
int vecchi[] = {1, 2, 3, 4};  
  
// nuovo array più lungo  
int nuovo[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};  
  
// copia tutti i vecchi elementi nel nuovo array  
System.arraycopy(vecchi, 0, nuovo, 0, vecchi.length);
```

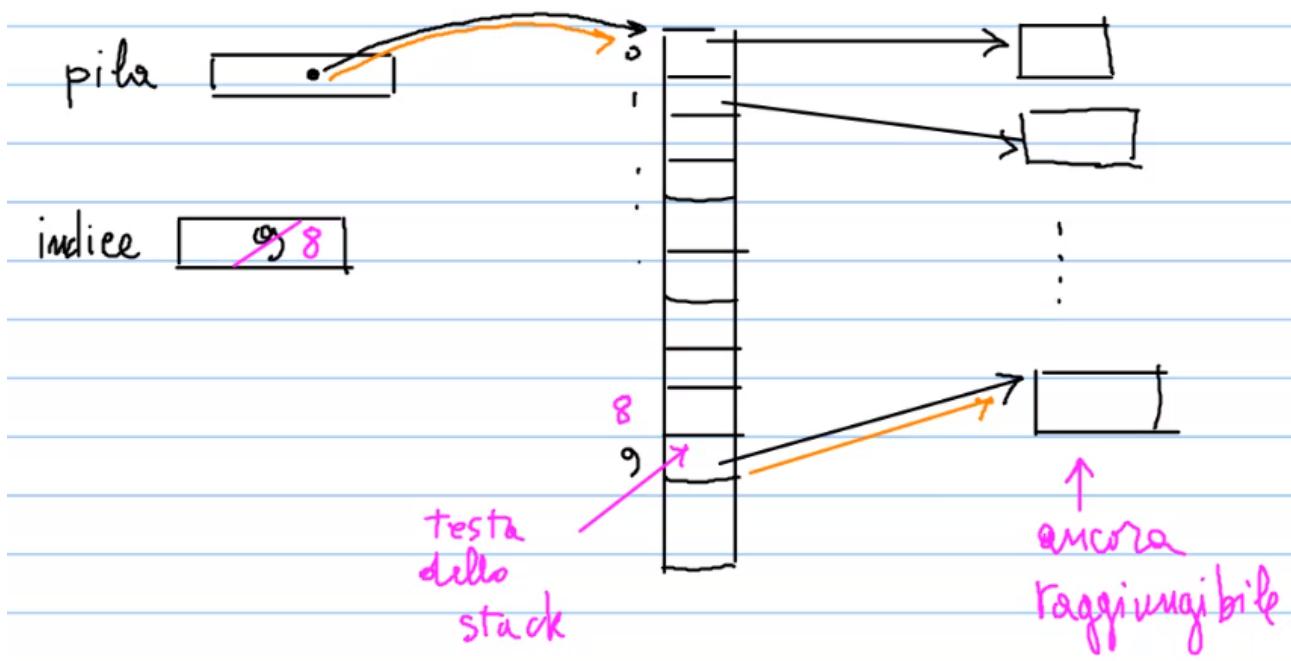


Attenzione: il metodo copia i valori contenuti negli elementi

dell'array. Nel caso di array di oggetti (o di array multidimensionali), ciò significa che vengono copiati i riferimenti agli oggetti, non vengono cioè create nuove copie di oggetti (vedi figura in alto a destra).

Aiutare il Garbage Collector

Il garbage collector non è onnipotente. Alcune volte ha bisogno di un piccolo "aiuto". Per esempio, supponiamo di dover operare su una pila usando il metodo: `public Object pop() { return pila[indice --]; }` In questo caso anche se cancello logicamente l'oggetto, esso rimane fisicamente lì; quindi il garbage collector non sa quando deallocare.



Se l'utilizzatore del metodo abbandona il valore di ritorno ricevuto, esso non sarà eleggibile per GC, finché il riferimento ad esso nell'array pila non sarà sovrascritto. Questo potrebbe richiedere tempi lunghissimi. Più correttamente quindi, per interrompere questa catena devo mettere a *null* la cella 9 dallo stack così da rendere quell'oggetto irraggiungibile, ad esempio con il seguente metodo:

```
public Object pop() {  
    Object valore = pila[indice];  
    pila[indice--] = null;  
    return valore;  
}
```

Questionario (soluzioni in fondo)

- 1) Un tipo di dato con segno ha un ugual numero di valori positivi e negativi.
- A. Vero
 - B. Falso
- 2) Scegliere gli identificatori legali tra questi:
- A. *StringaLunghissimaSenzaSignificato*
 - B. *\$int*
 - C. *Bytes*
 - D. *\$1*
 - E. *finals*
- 3) Quali delle seguenti segnature sono valide per il metodo main?
- A. *public static void main()*
 - B. *public static void main(String args[])*
 - C. *public void main(String[] arg)*
 - D. *public static void main(String[] args)*
 - E. *public static int main(String[] arg)*
- 4) Se in un file sorgente sono presenti tutti e tre gli elementi top-level, in quale ordine devono apparire?
- A. import, package, class.
 - B. class, import, package.
 - C. package per primo, l'ordine degli altri non importa.
 - D. package, import, class.
 - E. import per primo, l'ordine degli altri non importa.
- 5) Si consideri la seguente linea di codice: *int [] x = new int [25];* Dopo l'esecuzione, quali delle seguenti affermazioni sono vere?
- A. *x[0]* è 0.
 - B. *x* è indefinito
 - C. *x* è 0
 - D. *x[0]* è null
 - E. *x.length* è 25
- 6) Qual è l'output della seguente applicazione:
- ```
class D6 {
 public static void main (String args []) {
 Scatola s = new Scatola ();
 s.interno = 100;
 s.aumenta(s);
 System.out.println(s.interno);
 }
}

class Scatola {
 public int interno;
 public void aumenta(Scatola scatola) {
 scatola.interno++;
 }
}
```
- A. 0      B. 1      C. 100      D. 101

7) Qual è l'output della seguente applicazione:

```
class D7 {
 public static void main (String args[]) {
 double d = 12.3;
 Decremento dec = new Decremento();
 dec.decrementa(d);
 System.out.println(d);
 }
}

class Decremento {
 public void decrementa(double dec) {
 dec = dec - 1.0;
 }
}
```

- A. 0.0              B. -1.0              C. 12.3              D. 11.3

8) Come si può forzare la garbage collection di un oggetto?

- A. Il garbage collector non può essere forzato.
- B. Con una chiamata a `System.gc()`
- C. Con una chiamata a `System.gc()`, passando il riferimento all'oggetto
- D. Con una chiamata a `Runtime.gc()`
- E. Ponendo tutti i riferimenti a quell'oggetto a `null`

9) Qual è il range di valori per una variabile di tipo short?

- A. Dipende dall'hardware che ospita la JVM
- B.  $0 \dots 2^{16} - 1$
- C.  $0 \dots 2^{32} - 1$
- D.  $-2^{15} \dots 2^{15} - 1$
- E.  $-2^{31} \dots 2^{31} - 1$

10) Qual è il range di valori per una variabile di tipo byte?

- A. Dipende dall'hardware che ospita la JVM
- B.  $0 \dots 2^8 - 1$
- C.  $0 \dots 2^{16} - 1$
- D.  $-2^7 \dots 2^7 - 1$
- E.  $-2^{15} \dots 2^{15} - 1$

11) Quali sono i valori di x, a, b dopo l'esecuzione del codice:

```
int x, a = 6, b = 7;
x = (a++) + (b++);
```

- A.  $x = 15, a = 7, b = 8$
- B.  $x = 15, a = 6, b = 7$
- C.  $x = 13, a = 7, b = 8$
- D.  $x = 13, a = 6, b = 7$

12) Quali delle seguenti espressioni sono legali?

- A. `int x = 6; x = !x;`
- B. `int x = 6; if (!(x > 3)) {}`
- C. `int x = 6; x = ~x;`

13) Quali delle seguenti espressioni risultano in un valore positivo in x?

- A. `int x = -1; x = x >>> 5;`
- B. `int x = -1; x = x >>> 32;`
- C. `byte x = -1; x = x >>> 5;`
- D. `int x = -1; x = x >> 5;`

**14)** Quali delle seguenti espressioni sono legali?

- A. `String x = "Ciao"; int y = 7; x += y;`
- B. `String x = "Ciao"; int y = 7; if (x == y) {}`
- C. `String x = "Ciao"; int y = 7; x = x + y;`
- D. `String x = "Ciao"; int y = 7; y = y + x;`
- E. `String x = null;`  
`int y = (x != null) && (x.length() > 0) ? x.length() : 0;`

**15)** Qual è il risultato dell'esecuzione del seguente codice?

```
public class Xor {
 public static void main(String args[]) {
 byte b = 10; // 00001010 binario
 byte c = 15; // 00001111 binario
 b = (byte) (b^c);
 System.out.println("b vale " + b);
 }
}
```

- A. `b` vale 10
- B. `b` vale 5
- C. `b` vale 250
- D. `b` vale 245

**16)** Qual è il risultato della compilazione ed esecuzione del seguente codice?

```
1 public class Condizionale {
2 public static void main(String args[]) {
3 int x = 4;
4 System.out.Println("Il valore è " +
5 ((x > 4) ? 99.99 : 9));
6 }
7 }
```

- A. Il valore è 99.99
- B. Il valore è 9
- C. Il valore è 9.0
- D. Un errore di compilazione alla linea 5

**17)** Qual è l'output del seguente frammento di codice?

```
int x = 3;
int y = -10;
System.out.println(y * x);
```

- A. 0
- B. 1
- C. -1
- D. -3

**18)** Qual è l'output del seguente frammento di codice?

```
int x = 1;
String [] nomi = {"Mario", "Anna", "Carlo"};
nomi[--x] += ".";
for (int i = 0; i < nomi.length; i++) {
 System.out.println(nomi[i]);
}
```

- A. L'output include `Mario.` con un punto finale
- B. L'output include `Anna.` con un punto finale
- C. L'output include `Carlo.` con un punto finale
- D. Nessun nome stampato ha il punto finale.
- E. Viene lanciata l'eccezione `ArrayIndexOutOfBoundsException`

**19) Quali linee fanno parte dell'output del seguente codice?**

```
for (int i = 0; i < 2; i++) {
 for (int j = 0; j < 3; j++) {
 if (i == j) {
 continue;
 }
 System.out.println("i = " + i + " j = " + j);
 }
}
```

- A.  $i = 0 \ j = 0$
- B.  $i = 0 \ j = 1$
- C.  $i = 0 \ j = 2$
- D.  $i = 1 \ j = 0$
- E.  $i = 1 \ j = 1$
- F.  $i = 1 \ j = 2$

**20) Quali linee fanno parte dell'output del seguente codice?**

```
esterno : for (int i = 0; i < 2; i++) {
 for (int j = 0; j < 3; j++) {
 if (i == j) {
 continue esterno;
 }
 System.out.println("i = " + i + " j = " + j);
 }
}
```

- A.  $i = 0 \ j = 0$
- B.  $i = 0 \ j = 1$
- C.  $i = 0 \ j = 2$
- D.  $i = 1 \ j = 0$
- E.  $i = 1 \ j = 1$
- F.  $i = 1 \ j = 2$

**21) Quali tra queste sono costruzioni legali di loop?**

- A. 

```
while (int i < 7) {
 i++;
 System.out.println("i = " + i);
}
```
- B. 

```
int i =3;
while (i) {
 System.out.println("i = " + i);
}
```
- C. 

```
int j = 0;
for (int k = 0; j + k != 10; j ++, k++) {
 System.out.println("j = " + j + "k = " + k);
}
```
- D. 

```
int j = 0;
do {
 System.out.println ("j = " + j++);
 if (j == 3) {
 continue loop;
 }
} while (j < 10);
```

**22)** Qual è l'output di questo frammento di codice?

```
int x = 0, y = 4, z = 5;
if (x > 2) {
 if (y < 5) {
 System.out.println(" uno ");
 }
 else {
 System.out.println(" due ");
 }
}
else if (z > 5) {
 System.out.println(" tre ");
}
else {
 System.out.println(" quattro ");
}
```

- A. uno      B. due      C. tre      D. quattro

**23)** Dato il codice successivo, quale dei seguenti enunciati è vero?

```
1 int j = 2;
2 switch (j) {
3 case 2:
4 System.out.print("2");
5 case 2 + 1:
6 System.out.print("3");
7 break;
8 default :
9 System.out.print(j);
10 break;
11 }
12 System.out.println();
```

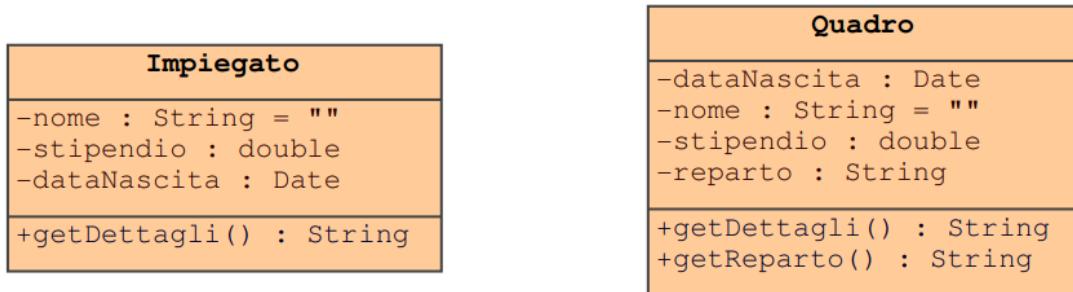
- A. Il codice è illegale a causa dell'espressione alla linea 5  
B. I tipi accettabili per la variabile di controllo di uno *switch* sono *byte*, *short*, *int*, *long*  
C. L'output è 2  
D. L'output è 23  
E. L'output è 232

**SOLUZIONI:** 1) B perché lo 0 “si mangia” un numero positivo. 2) Sono tutti legali 3) B e D 4) D 5) E 6) D 7) C infatti abbiamo un tipo primitivo quindi non modifichiamo il parametro attuale, IN per copia 8) A, il garbage collector opera tutte le volte che il programma è in attesa di qualcosa, quindi posso solo permettere ad una variabile di essere liberata ma non decido io quando 9) D (short ha 16 bit) 10) D 11) C 12) B e C, il not booleano non può essere usato per i tipi interi. 13) A (la struttura di bit dell'intero -1 è “tutti uno” di conseguenza (A) introduce a sinistra 5 zeri e trasforma l'intero in numero positivo; (B) introduce 32 modulo 32 = 0 zeri ed il numero resta invariato; (D) non modifica il bit di segno; (C) è un'operazione illegale poiché l'espressione *x >>> 5* è di tipo *int*, non compatibile per assegnazione con un *byte*. 14) A, C ed E ((A) e (C) sono identiche e producono in *x* il valore “ciao7”; (B) è illegale nel test (*x == y*); (D) + illegale poiché l'operatore di concatenazione produrrebbe la String di valore “7Ciao”, incompatibile per assegnazione con un *int*; (E) non da errore grazie al cortocircuito. 15) B (00001010) 16) C perché sono stati promossi tutti a *double* 17) C 18) A 19) B,C,D e F 20) D 21) Solo C (A non è legale perché i non è inizializzata, infatti il while non permette una dichiarazione interna, ed una dichiarazione non permette una condizione booleana; B non legale perché java richiede un boolean come condizione; D non è legale poiché l'etichetta loop non è stata definita) 22) D 23) D (La B è falsa perché i long sono troppo grandi, lo switch non le accetta)

## 38. Ereditarietà

### Specializzazione

Prendiamo come esempio le seguenti classi *Impiegato* e *Quadro*:



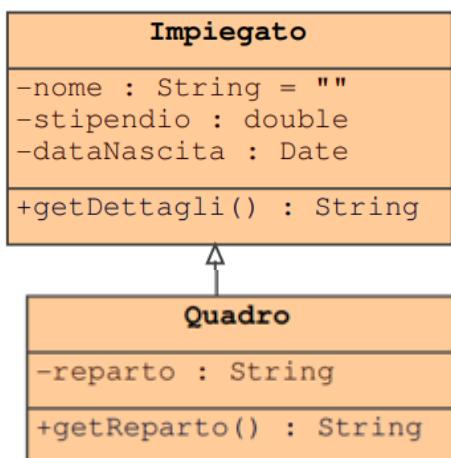
```
public class Impiegato {
 private String nome = "";
 private double stipendio;
 private Date dataNascita;

 public String getDettagli() {
 ...
 }
}
```

```
public class Quadro {
 private String nome = "";
 private double stipendio;
 private Date dataNascita;
 private String reparto;

 public String getDettagli() {...}
 public String getReparto() {
 return reparto;
 }
}
```

Si noti che le due classi hanno molte parti in comune, più precisamente *quadro* aggiunge ad *impiegato* solo un dato ed un metodo in più, quindi si possono ridefinire le precedenti classi nel seguente modo:



```
public class Impiegato {
 private String nome = "";
 private double stipendio;
 private Date dataNascita;

 public String getDettagli() {
 ...
 }
}

public class Quadro
extends Impiegato {
 private String reparto;
 public String getReparto() {
 return reparto;
 }
}
```

Questa definizione è più corretta dal punto di vista semantico, ma bisogna stare attenti. La parola *extends* non crea una classe separata di *impiegato* bensì rende gli oggetti di tipo *impiegato* anche oggetti di tipo *quadro*, l'ereditarietà inoltre specifica che la classe *quadro* sia una sottoclasse di *impiegato*.

La specializzazione garantisce inoltre i seguenti enunciati:

- Una sottoclasse eredita tutti i metodi e le variabili della superclasse.
- Una sottoclasse non eredita i costruttori della superclasse.
- I due modi, esclusivi uno con l'altro, per includere i costruttori in una classe sono:
  - usare il costruttore di default (senza parametri);
  - scrivere uno o più costruttori esplicativi.

## Polimorfismo

La strutturazione precedente crea un polimorfismo (gli oggetti di uno stesso tipo hanno forme diverse).

Un oggetto ha una sola forma, quindi se creo un oggetto *quadro* esso sarà *quadro* fino alla deallocazione. Il polimorfismo è l'abilità di riferirsi con la medesima variabile ad oggetti con forme diverse (credendo, però, di riferirsi ad un oggetto di una particolare forma ed ignorando, cioè, la reale forma dell'oggetto).

Poiché si può dire sia: *Impiegato impiegato = new Impiegato();* sia: *Impiegato impiegato = new Quadro();* la stessa variabile *impiegato* può essere usata per riferirsi ai due oggetti distinti; l'utilizzatore della variabile in entrambi i casi crederà di riferirsi ad una istanza di *Impiegato*.

Non sarà, pertanto, mai legale scrivere: *String reparto = impiegato getReparto();* anche se *impiegato* fosse in realtà un riferimento ad una istanza di *Quadro*.

Per collezioni omogenee si intendono oggetti di una stessa classe, mentre per collezioni eterogenee oggetti di classi diverse (immagine a sinistra collezione omogenea, a destra eterogenea)

```
MiaData [] data = new MiaData[2];
data[0] = new MiaData(5, 5, 2006);
data[1] = new MiaData(25, 12, 2006);
```

```
Impiegato [] staff = new Impiegato [100];
staff [0] = new Impiegato();
staff [1] = new Impiegato();
staff [2] = new Quadro();
```

## Argomenti polimorfici e operatore instance of

Si possono costruire metodi che accettino come parametro un riferimento “generico” ed operare in modo automatico su un più vasto insieme di oggetti. In questo caso è il metodo ad ignorare la reale natura (forma) dell'oggetto che gli viene specificato come parametro attuale. Per esempio, poiché *Quadro* è un *Impiegato*:

```
// Nella classe A
public double getIrpef (Impiegato i) {
 ...
}
```

```
// In un'altra classe:
A a = new A();
Quadro q = new Quadro();

double irpef = a.getIrpef(q)
```

Poiché è lecito scambiarsi oggetti usando riferimenti ai loro antenati (nella gerarchia ad albero), potrebbe essere a volte necessario sapere esattamente la forma dell'oggetto con cui si ha a che fare. Un modo per scoprire se l'oggetto è membro di una sottoclasse si usa l'operatore *instanceof*.

Per esempio, supponiamo che vi sia una gerarchia di classi:

```
public class Impiegato extends Object // ridondante
public class Quadro extends Impiegato
public class Segretario extends Impiegato
```

Se si riceve un oggetto usando un riferimento ad *Impiegato*, esso potrebbe essere anche realmente un *Quadro* o un *Segretario*. Per saperlo:

```
public void faQualcosa (Impiegato i) {
 if (i instanceof Quadro) {
 // elabora un Quadro
 }
 else if (i instanceof Segretario) {
 // elabora un Segretario
 }
 else {
 // elabora un Impiegato qualunque
 }
}
```

Generalmente l'utilizzo di *instanceof* è inutile o addirittura dannoso (sono rari i casi in cui l'operatore è necessario), dunque se si arriva a doverlo usare, di solito, è segno di scorretta programmazione poiché è il metodo dinamico di specie (Dynamic Method Dispatch) ad occuparsi automaticamente dell'elaborazione delle classi in base alla gerarchia.

## Casting

Nell'eventualità che si riceva un riferimento ad un oggetto e che (usando *instanceof*) si sappia che l'oggetto è realmente di una sottoclasse, si dovrebbe comunque usare quell'oggetto come se fosse della superclasse. Infatti, la formalizzazione di polimorfismo rende invisibili tutte le specializzazioni proprie di quell'oggetto.

Per rendere visibili tutte le caratteristiche dell'oggetto occorre fare un "cast" esplicito:

```
public void faQualcosa (Impiegato i) {
 if (i instanceof Quadro) {
 Quadro q = (Quadro) i;
 System.out.println ("Questo è il coordinatore del reparto " + q.getReparto());
 }
 ...
}
```

Se non ci fosse il cast, il metodo *getReparto* sarebbe stato inaccessibile al compilatore.

Ogni tentativo di effettuare un cast di riferimenti ad oggetti è soggetto a regole precise. A tale scopo, siano B e C due tipi:

- Gli "upcast" (si trasforma il tipo di un oggetto da sottoclasse a superclasse), sono sempre permessi e, infatti, non richiedono l'operatore di cast. Essi sono realizzati con una semplice assegnazione. Il codice: *C c; ... B b = c;* è corretto a patto che C sia sottoclasse di B, indipendentemente dall'oggetto a cui si riferisce c
- Per i "downcast" (da superclasse a sottoclasse) il compilatore controlla che l'operazione sia almeno possibile: la classe del riferimento di destinazione deve essere una sottoclasse del riferimento di origine. Il codice: *B b; ... C c = (C)b;* è corretto in compilazione se C è sottoclasse di B. I downcast sono rischiosi poiché non necessariamente una superclasse contiene un elemento della sottoclasse di conseguenza il cast deve essere esplicito.

Una volta che il compilatore abbia ammesso l'operazione, essa sarà infine controllata a runtime, per verificare che il tipo dell'oggetto riferito sia compatibile con il tipo del riferimento di destinazione.

Se, per esempio, si omettesse il controllo con *instanceof* e si realizzasse un downcast con un oggetto che non è realmente del tipo giusto (quello di destinazione, per intenderci), verrà lanciata una eccezione in esecuzione.

## 39. Altre relazioni tra classi

### Composizioni

La composizione implica una relazione a vita, quindi se il "tutto" è creato, anche le parti sono create, mentre quando il tutto muore, anche le parti muoiono. Le parti dell'oggetto composito sono esclusive di quell'oggetto, inoltre il costruttore per il tutto deve costruire anche le parti che lo compongono per mantenere la sue esclusività, ad esempio:

```
public class Automobile {
 private Motore motore;
 // altri attributi

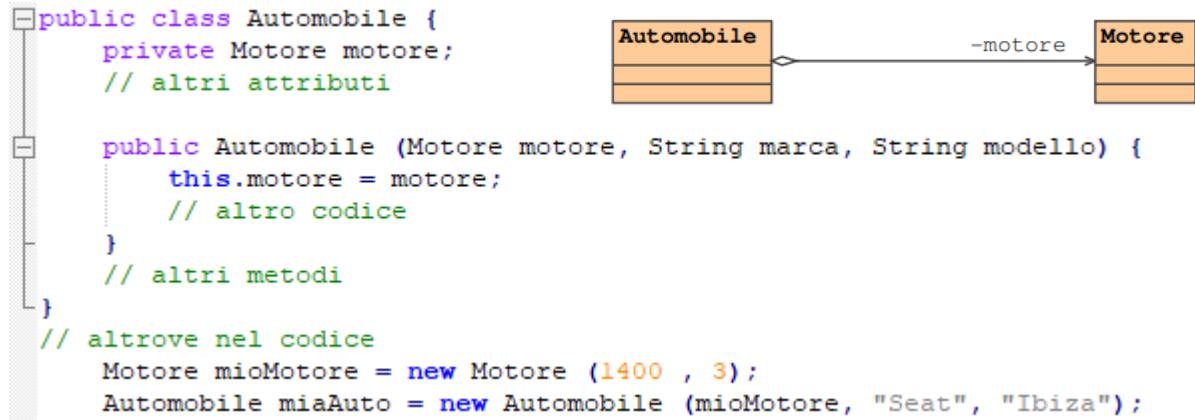
 public Automobile (int cilindrata, int numeroCilindri, String marca, String modello) {
 motore = new Motore (cilindrata, numeroCilindri);
 // altro codice
 }

 // altri metodi
}
```



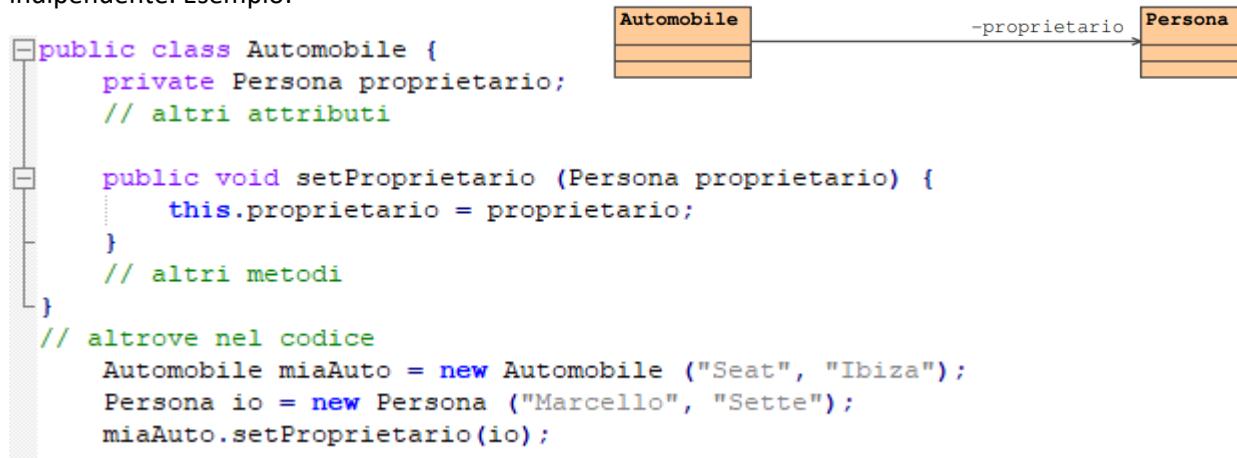
## Aggregazioni

Le parti possono essere oggetti di più parti, infatti nella aggregazione le parti sono create altrove e vengono inglobate nell'aggregante nel momento in cui esso viene costruito. Esiste un loro riferimento anche al di fuori dell'aggregante: in questo modo esse non cessano (forse, dipende da come si è costruito la parte) di esistere dopo la morte di esso. Esempio:



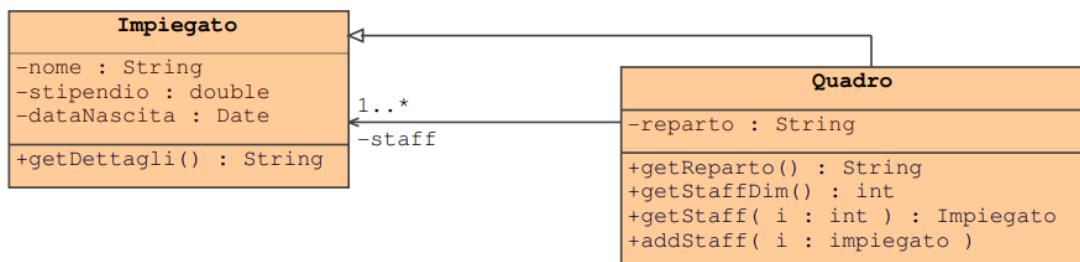
## Associazioni

Nella semplice associazione le vite degli oggetti, sia pure correlate, esistono in modo totalmente indipendente. Esempio:



## Molteplicità

La molteplicità è la notazione delle relazioni che mi dice il numero di oggetti che corrispondono ad un altro oggetto ad esso associato. Per l'implementazione di questo tipo di relazione ovviamente non basta un singolo puntatore ma ho bisogno di un numero variabile di puntatori. Esempio:



```

public class Quadro extends Impiegato {
 private String reparto = " ";
 private Impiegato [] staff = new Impiegato[20];
 private int staffDim = 0;

 public void getReparto() { return reparto }
 public int getStaffDim() { return staffDim }
 public Impiegato getStaff(i : int) { return staff[i] }
 public void addStaff(i : Impiegato) { staff[staffDim++] = i }
}

```

## 40. Overloading e Overriding

### Overloading di metodi

Si ha overloading quando metodi nella **stessa** classe hanno lo **stesso** nome e svolgono lo stesso compito ma hanno diversi argomenti. La lista degli argomenti deve essere diversa, perché essa sola è usata per distinguere i metodi, mentre il tipo di ritorno può essere diverso visto che non è usato per distinguere i metodi (cioè, metodi con uguale nome e lista di parametri, ma diverso tipo di ritorno sono indistinguibili ed il compilatore segnala l'errore). Per esempio: `public void println(int i);`   `public void println(float f);`

### Overloading di costruttori

Le stesse regole descritte nell'overloading di metodi si attuano anche con l'overloading di costruttori, in più c'è la possibilità di usare il riferimento `this`, nella prima linea di un costruttore, per invocare un altro costruttore.

```

public class Impiegato {
 private static final double STIPENDIO_BASE = 15000.00;
 private String nome ; private double stipendio ;
 private Date dataNascita ;

 public Impiegato (String nome, double stip, Date nasc) {
 this.nome = nome; stipendio = stip; dataNascita = nasc;
 }
 public Impiegato (String nome, double stip) {
 this(nome, stip, null);
 }
 public Impiegato (String nome, Date nasc) {
 this(nome, STIPENDIO_BASE, nasc);
 }
 public Impiegato (String nome) {
 this(nome, STIPENDIO_BASE);
 }
 // Altro codice
}

```

L'utilizzo di `this` è per evitare di scrivere la classe nel modo di figura a destra che è errato perché i costruttori possono comparire solo in istruzioni di tipo `new`.

```

public Impiegato (String n, double st) {
 Impiegato (n, St, null);
}

```

### Overriding di metodi

Si parla di overriding quando una sottoclasse sovrappone un metodo ereditato e visibile, a patto che, rispetto al metodo della superclasse, il nuovo metodo abbia lo stesso nome, lo stesso tipo di ritorno e la stessa lista di argomenti ed una visibilità non inferiore (vedremo poi).

```

public class Impiegato {
 protected String nome;
 protected double stipendio;
 protected Date dataNascita;

 public String getDettagli() {
 return "Nome: " + nome + "\n" + "Stipendio: " + stipendio;
 }
}
public class Quadro extends Impiegato {
 protected String reparto;

 public String getDettagli() {
 return "Nome: " + nome + "\n" + "Stipendio: " + stipendio + "\n" + "Reparto: " + reparto;
 }
}

```

In riferimento all'esempio precedente, è chiaro quali metodi sono invocati in:

```

Impiegato i = new Impiegato();
Quadro q = new Quadro();
System.out.println(i.getDettagli());
System.out.println(q.getDettagli());

```

Ma può essere meno chiaro quale metodo viene invocato nelle seguenti istruzioni:

```

Impiegato i = new Quadro();
System.out.println(i.getDettagli());

```

Viene chiamato il metodo dell'oggetto quadro, poiché seguendo il puntatore i si trova un oggetto creato come tipo quadro, anche se la variabile è di tipo impiegato. Si ottiene il comportamento del tipo che assumerà la variabile in esecuzione e non il comportamento del tipo che la variabile ha in compilazione (invocazione virtuale di metodo).

Questo meccanismo prende il nome di Dynamic Method Dispatch (in java) conosciuto negli altri linguaggi come Dynamic Binding o Late Binding. Nel linguaggio Java se non si danno indicazioni esplicite il binding è dinamico (si usa *static* per binding statico) opposto a C++ dove se non si hanno indicazioni il binding è statico (perché costa meno), mentre per quello dinamico bisogna usare la parola chiave *virtual*.

Il binding dinamico viene fatto in base al contenuto della variabile, mentre quello statico in base al tipo della variabile. L'invocazione virtuale dei metodi (Dynamic Method Dispatch) è la massima rivelazione del polimorfismo.

Riguardo al vincolo sulla non inferiore visibilità, il codice seguente non viene compilato (a dispetto del fatto che la vera risoluzione di tipo avviene in esecuzione):

```

public class Padre {
 public void faQualcosa() { }
}
public class Figlio extends Padre {
 private void faQualcosa() { }
}
public class UsaEntrambi {
 public void faAltro () {
 Padre p1 = new Padre();
 Padre p2 = new Figlio();
 p1.faQualcosa();
 p2.faQualcosa();
 }
}

```

## 41. Costruzione di oggetti

### La parola chiave **super**

- *super* è usata in una classe per riferirsi alla superclasse.
- *super* è usata per riferirsi ai membri della superclasse (attributi e metodi) con la dot notation.
- I membri della superclasse accessibili con *super* sono anche quelli che la superclasse ha ereditato e non solo quelli esplicitamente definiti in essa.
- Non si può usare *super.super.membro* per accedere a membri di classi di livello superiore alla prima superclasse.

### Invocazione di un *super*-costruttore

- Si può invocare uno dei costruttori della superclasse usando *super(...)* come prima linea nel proprio costruttore, con analoga sintassi di quella del proprio costruttore (e.g. *this(...)*).
- Se, come prima linea nel proprio costruttore, non c'è né *this*, né *super*, il compilatore pone una chiamata implicita a *super()*, cioè al costruttore di default della superclasse.
  - In questo caso, se non esiste il costruttore di default nella superclasse, il compilatore genera un errore.

```
public class Quadro
 extends Impiegato {
private String reparto;

public Quadro(String nome,
 double stipendio,
 String reparto) {
 super(nome, stipendio);
 this.reparto = reparto;
}
public Quadro(String nome,
 String reparto) {
 super(nome);
 this.reparto = reparto;
}
public Quadro(String reparto) {
 this.reparto = reparto;
 // errore: assenza del
 // costruttore Impiegato()
 // nella superclasse
}
```

### Costruzione ed inizializzazione di oggetti

Come visto in precedenza, quando ho una chiamata `new` viene allocata nello heap la memoria per il nuovo oggetto e sono inizializzate le variabili di istanza ai valori impliciti di default. Per ogni successiva chiamata ad un costruttore, cominciando dal costruttore individuato per primo, viene iterata questa sequenza di passi:

Esempio: Esecuzione di `new Quadro("M 7", "vendite")`

0. Inizializzazione base (`new...`).
  - 0.1. Allocazione memoria per un oggetto *Quadro*.
  - 0.2. Inizializzazione di reparto al valore di default *null*.
1. Esecuzione del costruttore:  
*Quadro("M 7", "vendite")*.
  - 1.1. Inizializzazione dei parametri del costruttore: *n = "M 7"*, *r = "vendite"*.
  - 1.2. Non c'è chiamata a *this(...)*.
  - 1.3. Invocazione di *super("M 7")* come istanza di *Impiegato(String n)*.
    - 1.3.1. Inizializzazione del parametro del costruttore: *n = "M 7"*
    - 1.3.2. Esecuzione di *this("M 7", null)* come istanza di *Impiegato(String n, Date d)*.
      - 1.3.2.1. Inizializzazione dei parametri del costruttore: *n = "M 7"*, *d = null*.
      - 1.3.2.2. Nessuna chiamata esplicita a *this(...)*.
      - 1.3.2.3. Nessuna chiamata esplicita a *super(...)*:

```
public class Impiegato {
 private String nome;
 private double stipendio = 15000.00;
 private Date dataNascita;

 public Impiegato(String n, Date d) {
 nome = n;
 dataNascita = d;
 }
 public Impiegato(String n) {
 this(n, null);
 }

public class Quadro extends Impiegato {
 private String reparto;
 public Quadro (String n , String r) {
 super(n);
 reparto = r;
 }
}
```

viene eseguito implicitamente `super()` come istanza di `Object()` (la classe `Object` è la radice di tutti gli oggetti).

- 1.3.2.3.1. Nessuna inizializzazione di parametri.
  - 1.3.2.3.2. Nessuna chiamata a `this(...)`.
  - 1.3.2.3.3. Nessuna chiamata a `super(...)` (`Object` è la radice).
  - 1.3.2.3.4. Nessuna inizializzazione esplicita.
  - 1.3.2.3.5. Nessun corpo da eseguire.
- 1.3.2.4. Inizializzazione esplicita della variabile: `stipendio = 15000.00`.
- 1.3.2.5. Esecuzione del corpo del costruttore: `nome = "M 7", dataNascita = null`
- 1.3.3. Saltato
- 1.3.4. Saltato
- 1.3.5. Nessun corpo in `Impiegato(String n)`.
- 1.4. Nessuna inizializzazione esplicita per `Quadro`.
- 1.5. Esecuzione di `reparto="Vendite"`.

## 42. La classe `Object` e le classi wrapper.

### La classe `Object`

In Java la classe `Object` è la radice di tutte le classi, ovvero la classe più generale di tutte, quindi è l'unica a non avere una superclasse, inoltre una dichiarazione di classe senza la clausola `extends`, implicitamente usa “`extend Object`” e questo ci permette di sovrapporre numerosi metodi della classe `Object`. Questi metodi sono inseriti nella classe `Object` e quindi vengono per forza ereditati dalle sottoclassi.

### Il metodo `equals`

Mentre l'operatore `==` determina se due riferimenti sono identici (cioè se riferiscono allo stesso, unico, oggetto). Il metodo `equals` determina se le due variabili si riferiscono ad oggetti (anche distinti) appartenenti alla stessa classe di equivalenza, rispetto ad una particolare relazione di equivalenza definita dall'utente. La realizzazione di `equals` nella classe `Object` fa uso di `==`. Quindi le classi di equivalenza di `==` e di `equals` coincidono in `Object` (relazione di identità). Lo scopo del metodo è quello di essere sovrapposto nelle classi sviluppate dagli utenti, in modo da realizzare nuove relazioni di equivalenza. Alcune classi di sistema lo fanno già, per esempio la classe `String` realizza il proprio `equals`, confrontando le stringhe carattere per carattere. Viene raccomandato di sovrapporlo, insieme con `equals`, anche il metodo `hashCode` (usato usualmente per inserire l'oggetto in una tabella hash). Una decente, anche se povera, realizzazione potrebbe usare lo XOR bit a bit dei codici hash degli oggetti in esame:

```
public class Impiegato {
 private String nome;
 private MiaData nascita;
 private double stipendio;

 public Impiegato (String n, Date d, double s) {
 nome =n; dataNascita = d; stipendio =s;
 }
 public boolean equals(Object o) {
 boolean risultato = false;
 if ((o != null) && (o instanceof Impiegato)) {
 Impiegato i = (Impiegato) o;
 if (nome.equals(i.nome) && nascita.equals(i.nascita)) {
 risultato = true;
 }
 }
 return risultato;
 }
 public int hashCode() {
 return (nome.hashCode() ^ nascita.hashCode());
 }
}
```

## Il metodo `toString`

Questo metodo restituisce una rappresentazione String di un qualunque oggetto. È molto utile poiché è il metodo usato dalla print line e viene usato automaticamente durante le conversioni a stringhe:

```
Date now = new Date();
System.out.println(now); // rozzamente equivalente a System.out.println(now.ToString());
```

Occorre sovrapporre questo metodo quando si vuole fornire una rappresentazione di un oggetto in forma (umanamente) leggibile. I tipi primitivi sono rappresentati in forma di *String* usando il metodo statico `toString` della corrispondente classe "wrapper".

## Le classi "wrapper"

I tipi primitivi non sono oggetti. Se si vogliono manipolare come oggetti, è possibile avvolgere un singolo valore con un opportuno oggetto, cosiddetto "wrapper". La corrispondenza è la seguente:

| Tipo primitivo | boolean | byte | char      | short | int     | long | float | double |
|----------------|---------|------|-----------|-------|---------|------|-------|--------|
| Classe wrapper | Boolean | Byte | Character | Short | Integer | Long | Float | Double |

Si può costruire un oggetto wrapper, passando il relativo valore al costruttore appropriato. Per esempio:

```
int i = 500;
Integer indice = new Integer(i);
int x = indice.intValue();
```

Il valore da avvolgere può essere passato anche in una rappresentazione sotto forma di *String*. Il valore avvolto può essere estratto usando il metodo opportuno ...Value(). Le classi wrapper sono utili quando si vogliono convertire i tipi primitivi. Per esempio:

```
int x = Integer.valueOf(str).intValue();
int y = Integer.parseInt(str);
```

## 43. Modificatori

### Generalità

I modificatori sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi. Un gruppo di modificatori, detti di accesso, specificano a quali classi è permesso usare quella caratteristica. Altri modificatori possono essere usati, in combinazione con i precedenti, per qualificare quella caratteristica.

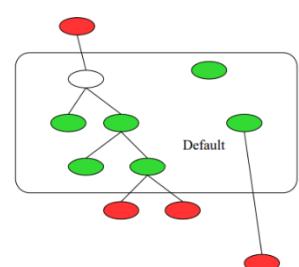
### Modificatori di accesso

Una *caratteristica* di una classe può essere la classe stessa; un attributo (variabile) della classe oppure un metodo o un costruttore della classe. Le sole variabili che possono essere controllate con i modificatori di accesso sono gli attributi (variabili di heap) e non le variabili dei metodi (variabili di stack): una variabile di un metodo può essere vista solo dal metodo in cui si trova.

Una caratteristica può avere al più un modifikatore di accesso, mentre, se non è presente nessun modifikatore, si intende accesso di default (non è una parola riservata) "accesso consentito dallo stesso pacchetto in cui è situata quella caratteristica".

I modificatori di accesso sono:

- *public*
  - È il modifikatore più generoso. Accesso consentito a tutte le altre classi.
  - Attenzione: l'accesso ad un attributo o un metodo public è subordinato all'accesso alla classe che lo contiene



- *protected*
    - Accesso consentito a tutte le classi nello stesso pacchetto
    - Accesso consentito a tutte le sottoclassi in pacchetti diversi, ma solo per ereditarietà: una sottoclasse in un altro pacchetto può accedere ad un membro *protected* nella superclasse solo attraverso un riferimento ad un oggetto del proprio tipo (anche all'oggetto corrente, mediante *this*, o, esplicitamente, alla sua parte ereditata, mediante *super*) o di un sottotipo.
  - *private*
    - È il modificatore meno generoso. Può essere usato solo per attributi o metodi, non per classi top-level
    - Le variabili *private* possono essere nascoste anche allo stesso oggetto che le possiede
    - Esempio:

```

class Complesso {
 private double reale , immag;
}
class SubComplesso extends Complesso {
 SubComplesso (double r , double i) {
 reale = r ; // Illegale
 }
}

```

      - La classe *SubComplesso* eredita gli attributi della superclasse, ma quegli attributi possono essere usati solo dal codice della classe *Complesso*.
      - Per quanto riguarda la classe *SubComplesso*, è come se quegli attributi non li avesse ereditati affatto.
- Tutto quanto detto è valido anche per metodi privati; i costruttori, invece, non sono mai ereditati.

Il solo modificatore di accesso o permesso per una classe top-level è *public*: non esistono classi top-level *protected* o *private*.

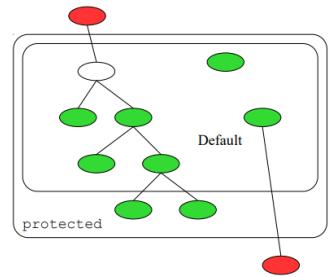
#### **Nota bene:**

- I metodi visibili di una superclasse non possono essere sovrapposti in una sottoclasse da altri meno visibili.

#### Altri modificatori

##### ***final*:**

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: una caratteristica *final* non può essere modificata.
  - Una classe *final* non può essere estesa, cioè non può avere sottoclassi.
  - Una variabile *final* è praticamente una costante, cioè può solo essere inizializzata. Attenzione: un riferimento *final* ad un oggetto non può essere modificato (cioè riassegnato ad un altro oggetto), ma l'oggetto a cui esso fa riferimento sì.
  - Un metodo *final* non può essere sovrapposto in una sottoclasse.
  - Non ha senso dire *final* ad un costruttore, perché esso non è mai ereditato dalle sottoclassi, quindi mai sovrapponibile.



### ***abstract:***

- Si applica solo a classi e a metodi. Un metodo abstract non possiede corpo (";" invece di "{...}"): 

```
abstract void getValore();
```
- Una classe può essere marcata abstract. In questo caso il compilatore suppone che essa contenga (anche per ereditarietà) metodi abstract: essa non potrà essere istanziata, anche se non contiene affatto metodi abstract.
- Una classe dove essere marcata abstract se:
  - contiene almeno un metodo abstract
  - eredita almeno un metodo abstract per il quale non fornisce una realizzazione
  - dichiara di implementare una interfaccia [vedremo in seguito], ma non fornisce una realizzazione di tutti i metodi di quell'interfaccia.
- In un certo senso, abstract è opposto a final: una classe final, per esempio, non può essere specializzata; una classe *abstract* esiste solo per essere specializzata.

### ***static:***

- Si applica ad attributi, metodi ed anche a blocchi di codice che non fanno parte di metodi.
- In generale, una caratteristica static appartiene alla classe, non alle singole istanze: essa è unica, indipendentemente dal numero (anche zero) di istanze di quella classe.
- Attributi:
  - L'inizializzazione di un attributo static avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {
 static int x = 0;
 Ecstatic () { x++; }
}
```
  - L'accesso ad un attributo static di una classe può avvenire (con la dot-notation) o partendo da un riferimento ad una istanza di quella classe, o partendo dal nome stesso della classe.

```
System.out.println(Ecstatic.x);
Ecstatic e = new Ecstatic();
e.x = 100;
Ecstatic.x = 100;
```
- Metodi:
  - Esistono nel momento in cui la classe viene caricata in memoria (anche senza istanze).
  - Non possono accedere a membri non static della stessa classe (perché potrebbero anche non esistere).
  - Non possono essere sovrapposti da metodi non-static.
  - Non possono sovrapporre metodi non-static.

### **Blocchi**

- È lecito che una classe contenga blocchi di codice marcati static.
  - Tali blocchi sono eseguiti una sola volta, nell'ordine in cui compaiono, quando la classe viene caricata in memoria e, ovviamente, possono accedere (come i metodi static) solo a caratteristiche static.

```
public class EsempioStatic {
 static double d = 1.23;
 static {
 System.out.println("Codice static: d = " + d++);
 }
 public static void main(String [] args) {
 System.out.println("main: d = " + d++);
 }
 static {
 System.out.println("Codice static: d = " + d++);
 }
}
```

### ***native:***

- Si applica solo a metodi.
- I metodi native non sono scritti in java ma è possibile fare interagire il codice java con questi tipi scritti in altri linguaggi, ovviamente bisogna scrivere solo il prototipo in Java.
- Come per abstract, native indica che il corpo di un metodo deve essere trovato altrove, in questo caso all'esterno della JVM, in una libreria di codice dipendente dalla architettura fisica.
- Usare questo tipo di metodi è molto complicato poiché richiede una serie di conoscenze specifiche.

### ***transient:***

- Si applica solo ad attributi.
- Indica che quell'attributo contiene informazioni sensibili e che, nel caso in cui lo stato interno dell'oggetto debba essere salvato (nel gergo "serializzato"), il valore di quell'attributo non dovrà essere considerato.
- Trasforma un oggetto in una sequenza di bite per essere trasmesse in rete (serializzazione) ma non è detto che contenga tutti i dettagli dell'oggetto, infatti potrei omettere dati che vengono inizializzati in seguito o magari omettere dei dati sensibili che non voglio vengano trasmessi in rete.

### ***synchronized:***

- Si applica solo a metodi o a blocchi anonimi.
- Controlla l'accesso al codice in programmi multi-thread.
- Un metodo synchronized garantisce che il programma venga eseguito da un thread alla volta, un eventuale nuovo thread che voglia eseguire lo stesso blocco già occupato viene messo in coda.

### ***volatile:***

- Si applica solo ad attributi.
- Sostanzialmente serve a spegnere delle procedure a rischio.
- Avverte il compilatore che quell'attributo può essere modificato in modo asincrono, in architetture multiprocessore.

## Sommario

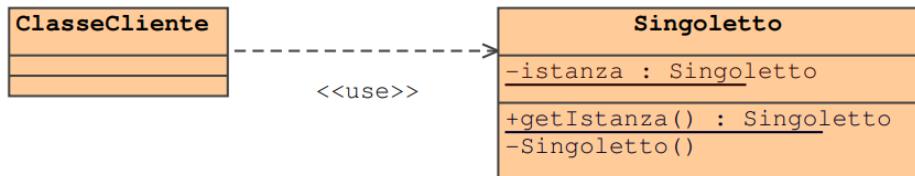
| Modificatore | Classe | Attributo | Metodo | Costruttore | Blocco |
|--------------|--------|-----------|--------|-------------|--------|
| public       | ✓      | ✓         | ✓      | ✓           |        |
| protected    |        | ✓         | ✓      | ✓           |        |
| "default"    | ✓      | ✓         | ✓      | ✓           | ✓      |
| private      |        | ✓         | ✓      | ✓           |        |
| final        | ✓      | ✓         | ✓      |             |        |
| abstract     | ✓      |           | ✓      |             |        |
| static       |        | ✓         | ✓      |             | ✓      |
| native       |        |           | ✓      |             |        |
| transient    |        | ✓         |        |             |        |
| volatile     |        | ✓         |        |             |        |
| synchronized |        |           | ✓      |             | ✓      |

Forme progettuali ricorrenti (Design Patterns) sono soluzioni a problemi ricorrenti nella progettazione OO.

## Singololetto

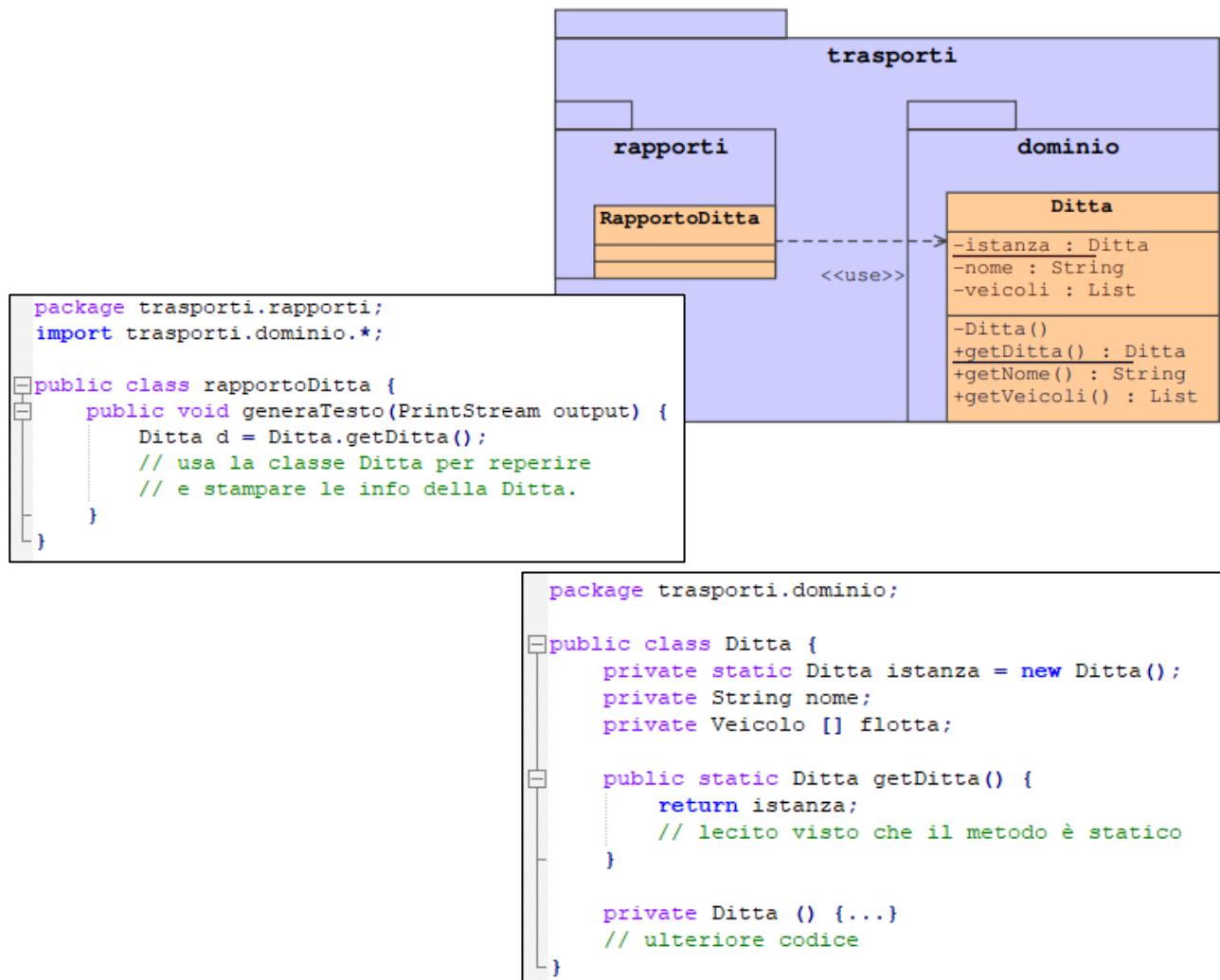
È uno dei requisiti che ricorre spesso nella programmazione oggetti, (in matematica viene chiamato singleton), ha lo scopo di evitare duplicazioni della classe, infatti se si scrivesse la classe nel solito modo, gli utilizzatori potrebbero creare istanze a volontà.

Lo scopo della forma progettuale Singleton è quello di assicurare il progettista della classe che esisterà sempre una ed una sola istanza di quella classe. La forma è la seguente:



Nota: In UML le componenti statici in una classe sono sottolineate. L'associazione tratteggiata `<<use>>` descrive non un attributo della *ClasseCliente* di tipo *Singololetto*, ma un uso diretto della classe *Singololetto*.

La classe singololetto avrà degli elementi specifici, come ad esempio un attributo privato che punti alla singola istanza della classe (di tipo singololetto), la sottolineatura indica che l'attributo sia statico. Un'altra caratteristica interessante è che visto che viene creata insieme alla classe è disponibile da subito anche se non si creano istanze di quell'oggetto (che si inizializza con il metodo, anch'esso statico `getIstante`). Si noti che persino il costruttore è privato e quindi può essere usato solo dalla classe Singololetto.



## Questionario con spiegazione

1) Quale dei seguenti frammenti viene correttamente compilato e stampa "Uguale" in esecuzione?

```
Integer x = new Integer (100);
Integer y = new Integer (100);
if (x == y) {
 System.out.println("Uguale");
}
```

A.

Se gli oggetti sono istanze l'== confronta i puntatori e quindi il test sarà falso poiché sono istanze diverse

```
int x = 100;
Integer y = new Integer (100);
if (x == y) {
 System.out.println("Uguale");
}
```

B.

Risulteranno diversi

```
int x = 100;
float y = 100.0F;
if (x == y) {
 System.out.println("Uguale");
}
```

C.

Viene stampato Uguale poiché l'int viene promosso a float e la condizione sarà vera

```
String x = new String("100");
String y = new String("100");
if (x == y) {
 System.out.println("Uguale");
}
```

D.

Analogo al caso A

```
String x = "100";
String y = "100";
if (x == y) {
 System.out.println("Uguale");
}
```

E.

Stampa "Uguale"

2) Quali delle seguenti dichiarazioni sono illegali?

A. *default String s;*

Default non è una parola chiave

B. *transient int i = 41;*

transient si applica solo ad attributi quindi la dichiarazione è lecita

C. *public final static native int w();*

Sono tutte parole chiavi applicabili a metodi e non in contrasto tra loro, quindi è legale

D. *abstract double d;*

Abstract si applica solo a metodi e classi

E. *abstract final double cosenoIperbolico();*

Abstract e final sono incompatibili poiché il primo mi dice di cercare l'implementazione del metodo nelle classi figli mentre final mi vieta l'overriding

**3) Quale delle seguenti affermazioni è vera?**

- A. Una classe abstract non può avere metodi final.

Una classe abstract non mi vieta di concretizzare un metodo al suo interno (anche final)

- B. Una classe final non può avere metodi abstract.

Una classe final non ha senso che abbia un metodo abstract poiché non avrà specializzazioni, inoltre un metodo abstract obbliga che la sua classe sia abstract (non vale il viceversa)

**4) Qual è la minima modifica che rende il seguente codice compilabile?**

```
1 final class Aaa {
2 int xxx;
3 void yyy() { xxx = 1; }
4 }
5
6 class Bbb extends Aaa {
7 final Aaa fref = new Aaa();
8 final void yyy() {
9 System.out.println("In yyy()");
10 fref.xxx = 12345;
11 }
12 }
```

- A. Alla linea 1, rimuovere final

Necessario altrimenti non potrei estendere la classe

- B. Alla linea 7, rimuovere final

Non necessario poiché è il puntatore ad essere immutabile e non l'attributo dell'istanza.

- C. Rimuovere la linea 10

Non necessario

- D. Alla linea 1 e 7, rimuovere final

Combinazione di A e B

- E. Nessuna modifica è necessaria

Va in contrasto con la A

**5) Riguardo al codice seguente, quale affermazione è vera?**

```
1 class Roba {
2 static int x = 10;
3 static { x += 5; }
4
5 public static void main (String [] args) {
6 System.out.println(" x =" + x);
7 }
8 static { x /= 5; }
9 }
```

- A. Le linee 3 e 8 non sono compilate, poiché mancano i nomi di metodi e i tipi di ritorno

I blocchi statici servono ad inizializzare variabili statiche come x

- B. La linea 8 non è compilata, poiché si può avere solo un blocco top-level static

Anche se sconsigliato metterli casualmente nel codice è possibile averne più di uno.

- C. Il codice viene compilato e l'esecuzione produce x = 10;

- D. Il codice viene compilato e l'esecuzione produce x = 15;

- E. Il codice viene compilato e l'esecuzione produce x = 3;

Essendo static viene modificato sempre lo stesso campo e i blocchi vengono eseguiti tutti una sola volta quando la classe è caricata in memoria dunque stampa 3.

6) Rispetto al codice seguente, quale affermazione è vera?

```
1 class A {
2 private static int x = 100;
3
4 public static void main (String [] args) {
5 A hsl = new A();
6 hsl.x++;
7 A hs2 = new A();
8 hs2.x++;
9 hsl = new A();
10 hsl.x++;
11 A.x++;
12 System.out.println("x = " + x);
13 }
14 }
```

- A. La linea 6 non compila, poiché è un riferimento static ad una variabile private
- B. La linea 11 non compila, poiché è un riferimento static ad una variabile private  
Essendo il metodo public definito internamente alla classe A non ci sono problemi di scoping
- C. Il programma viene compilato e stampa x = 102
- D. Il programma viene compilato e stampa x = 102
- E. Il programma viene compilato e stampa x = 104

Essendo static modifico sempre la stessa variabile, quindi sono validi tutti gli incrementi, x = 104

7) Dato il codice seguente:

```
1 class SuperC {
2 void unMetodo() {}
3 }
4
5 class SubC extends SuperC {
6 void unMetodo() {}
7 }
```

- A. Quali modificatori di accesso possono essere legalmente dati ad *unMetodo* alla linea 2, lasciando il resto del codice inalterato?  
Essendo un metodo a cui viene fatto overriding il metodo della sottoclasse deve essere più visibile (o avere la stessa visibilità) quindi alla linea 2, il metodo può essere private, visto che la sottoclasse viene sovrapposta da un metodo con visibilità di pacchetto.
- B. Quali modificatori di accesso possono essere legalmente dati ad *unMetodo* alla linea 6, lasciando il resto del codice inalterato?  
Alla linea 6, il metodo può essere protected oppure public, visto che sovrappone un metodo con visibilità di pacchetto.

8) Riguardo ai seguenti codici, quale affermazione è vera?

```
1 package abcd;
2
3 public class SupA {
4 protected static int count = 0;
5 public SupA() { count++; }
6 protected void f() {}
7 static int getCount() {
8 return count;
9 }
10 }
```

```
1 package abcd;
2
3 class A extends abcd.SupA {
4 public void f() {}
5 public int getCount() {
6 return count;
7 }
8 }
```

- A. La compilazione di A.java fallisce alla linea 4, poiché il metodo f() è protected nella superclasse e A è nello stesso pacchetto di SupA

Il metodo della superclasse viene ridefinito e la visibilità passa da protected a public, quindi si amplia la visibilità. Non ci sono problemi di compilazione

- B. La compilazione di A.java fallisce alla linea 4, poiché il metodo f() è protected nella superclasse e public nella sottoclassse.

Come sopra

- C. La compilazione di A.java fallisce alla linea 5, poiché il metodo getCount() è static nella superclasse e non può essere sovrapposto da un metodo non-static.

I metodi devono essere Static in entrambe le parti per avere un codice compilabile

- D. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo f() su una istanza di A

- E. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo getCount su una istanza di SupA

I codici non sono compilabili per la C

**9) Riguardo i codici seguenti, quale affermazione è vera?**

```

1 package abcd;
2
3 public class SupA {
4 protected static int count = 0;
5 public SupA() { count++; }
6 protected void f() {}
7 static int getCount() {
8 return count;
9 }
10}
11
12 package ab;
13
14 class A extends abcd . SupA {
15 A() { count++; }
16
17 public static void main (String [] args) {
18 System.out.print("Prima: " + count);
19 A a = new A ();
20 System.out.println("Dopo: " + count);
21 a.f();
22 }
23 }
```

- A. Il programma viene compilato e stampa: Prima: 0 Dopo: 2

- B. Il programma viene compilato e stampa: Prima: 0 Dopo: 1

L'inizializzazione esegue automaticamente il super e quindi viene incrementato dalla linea 5 e successivamente incrementato dalla linea 4 del package ab

- C. La compilazione di A fallisce alla linea 4.

Essendo count protected può essere utilizzata dalla sottoclassse

- D. La compilazione di A fallisce alla linea 10.

Essendo protetto viene ereditato e può essere usato

- E. Il programma viene compilato, ma viene lanciata una eccezione alla linea 10.

Non ci sono problemi a runtime

**10) Si consideri la classe seguente, Quali dei seguenti metodi può lecitamente essere inserito alla linea 4?**

```

1 public class Test1 {
2 public float unMetodo (float a, float b) {
3 }
4 }
5 }
```

- A. *public int unMetodo(int a, int b) {}*

- B. *public float unMetodo(float a, float b) {}*

- C. *public float unMetodo(float a, float b, int c) {}*

- D. *public float unMetodo(float c, float d) {}*

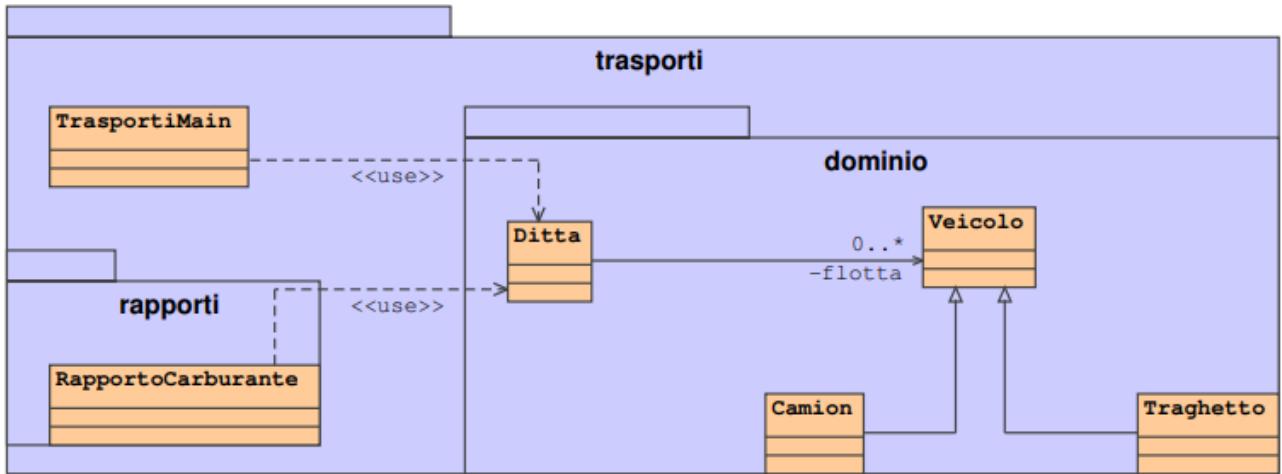
- E. *private float unMetodo(int a, int b, int c) {}*

La distinzione tra due metodi avviene solo se gli argomenti sono differenti in tipo e/o numero, di conseguenza non influiscono tipi di ritorno o i nomi (a, b, c, d) dei parametri

## 44. Classi astratte

### Introduzione

Supponiamo che si voglia un rapporto settimanale sul consumo dei veicoli di una ditta di trasporti. Il diagramma UML potrebbe essere questo:



Mentre, il codice che realizza le due classi esterne al dominio sia:

```
public class TrasportiMain {
 public static void main (String [] args) {
 Ditta d = Ditta.getDitta(); // Singoletto

 // popola la flotta di veicoli
 d.addVeicolo(new Camion (10000.0));
 d.addVeicolo(new Camion(15000.0));
 d.addVeicolo(new Traghetto(500000.0));
 d.addVeicolo(new Camion (9500.0));
 d.addVeicolo(new Traghetto(750000.0));

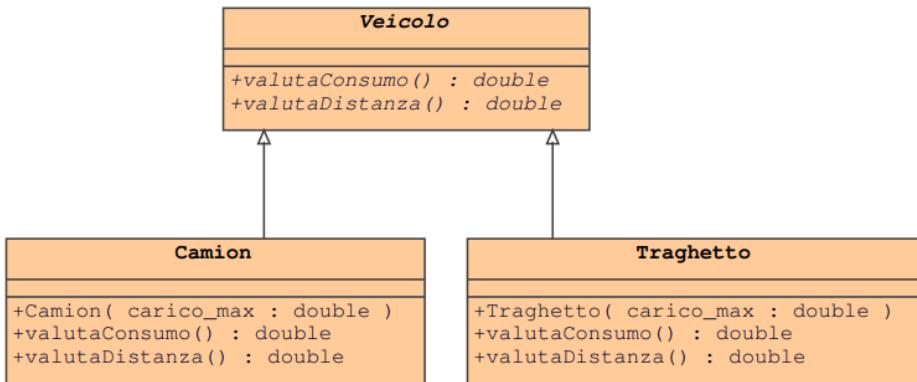
 RapportoCarburante rapporto = new RapportoCarburante();
 rapporto.generaTesto(System.out);
 }
}

public class RapportoCarburante {
 public void generaTesto(PrintStream output) {
 Ditta d = Ditta.getDitta(); // Singoletto
 Veicolo v;
 double carburante;
 double totale = 0.0;
 for (int i = 0; i < d.getDimFlotta(); i++) {
 v = d.getVeicolo(i);
 carburante = v.valutaConsumoPerKm() * v.valutaDistanza();
 output.println("Consumo del veicolo " + v.getNome() + ": " + carburante);
 totale += carburante;
 }
 output.println("Consumo totale: ", totale);
 }
}
```

### Problema A

La valutazione dei consumi tra i due tipi di veicoli potrebbe essere totalmente differente. Non ha senso che la classe *Veicolo* fornisca i due metodi *valutaDistanza()* e *valutaConsumoPerKm()*, ma le sue sottoclassi sì.

**Soluzione:** Java permette ad una superclasse di dichiarare un metodo, del quale non fornirà una realizzazione, quest'ultima infatti sarà implementata dalle sottoclassi. Tali metodi sono astratti e una classe con uno o più metodi astratti è una classe astratta.



In UML ricordiamo che metodi o classi astratte si rappresentano utilizzando il carattere corsivo. Java proibisce la costruzione di oggetti di una classe astratta. Tuttavia, classi astratte possono avere attributi, metodi concreti e costruttori. È buona norma rendere questi costruttori

protected piuttosto che public poiché non si devono avere istanze di una classe astratta.

```

public abstract class Veicolo {
 public abstract double valutaConsumoPerKm();
 public abstract double valutaDistanza();
}

public class Camion extends Veicolo {
 public Camion (double carico_max) {...}
 public double valutaConsumoPerKm () {
 // Restituisce il consumo/Km
 // per un certo modello di camion
 }
 public double valutaDistanza () {
 // Restituisce la distanza che deve
 // percorrere per i viaggi
 }
}

public class Traghetto extends Veicolo {
 public Traghetto (double carico_max) {...}
 public double valutaConsumoPerKm () {
 // Restituisce il consumo/Km
 // per un certo traghetto
 }
 public double valutaDistanza () {
 // Restituisce la distanza che deve
 // percorrere per i viaggi in mare
 }
}

```

## Problema B

Riprendiamo la classe *RapportoCarburante* (in particolare la linea evidenziata). Il calcolo del consumo di un veicolo non dovrebbe essere eseguito in questa classe, ma dovrebbe essere una responsabilità del veicolo stesso, cioè appartenere alla classe *Veicolo*. Ma nella classe *veicolo* non ci sono le realizzazioni dei metodi *valutaConsumoPerKm()* e *valutaDistanza()*. Ma non si riscontrerà nessun problema, poiché l'invocazione virtuale dei metodi serve proprio a questo. Deleghiamo, pertanto, ad un veicolo il calcolo del consumo:

```

public class RapportoCarburante {
 public void generaTesto(PrintStream output) {
 ...
 carburante = v.valutaConsumo();
 ...
 }
}

class Veicolo {
 -carico : double = 0
 -caricoMax : double = 0
 #valutaConsumoPerKm() : double
 #valutaDistanza() : double
 #Veicolo(carico_max : double)
 +getCarico() : double
 +getCaricoMax() : double
 +addScatola(peso : double)
 +valutaConsumo()
}

class Camion {
 +Camion(carico_max : double)
 +valutaConsumoPerKm() : double
 +valutaDistanza() : double
}

class Traghetto {
 +Traghetto(carico_max : double)
 +valutaConsumoPerKm() : double
 +valutaDistanza() : double
}

```

La tecnica di soluzione consiste nel porre in una classe astratta un metodo concreto (nel nostro caso `valutaConsumo()`) che utilizzi metodi astratti nella stessa classe. Poiché tale tecnica è assai ricorrente, essa viene comunemente indicata come *Metodo Sagoma (Template Method Design Pattern)*.

## 45. Interfacce

### Generalità

Una “interfaccia” è un contratto tra il codice cliente e la classe che “implementa” quell’interfaccia. Sono come classi completamente astratte dove l’unica cosa specificata sono signature di metodi e volendo delle costanti che servono come parametri per i metodi appartenenti all’interfaccia. In Java è una formalizzazione di un tale contratto in cui tutti i metodi non contengono realizzazioni.

Una interfaccia può essere implementata anche da molte classi non correlate (non è necessario che queste classi si trovano sullo stesso package o addirittura senza relazioni di sottoclasse o superclasse). Un tipo di dato così poco specificato ha lo scopo di “cucire” insieme pezzi di software in maniera uniforme.

Un interfaccia può estendere altre interfacce (ereditarietà multiple). La sintassi è la seguente:

```
<class_declarator> ::=
 <modifier> class <name> [extends <superclass>]
 [implements <interface> [, <interface>]*] {
 <class_body>
 }

<interface_declarator> ::=
 <modifier> interface <name>
 [extends <interface> [, <interface>]*] {
 <interface_body>
 }
```

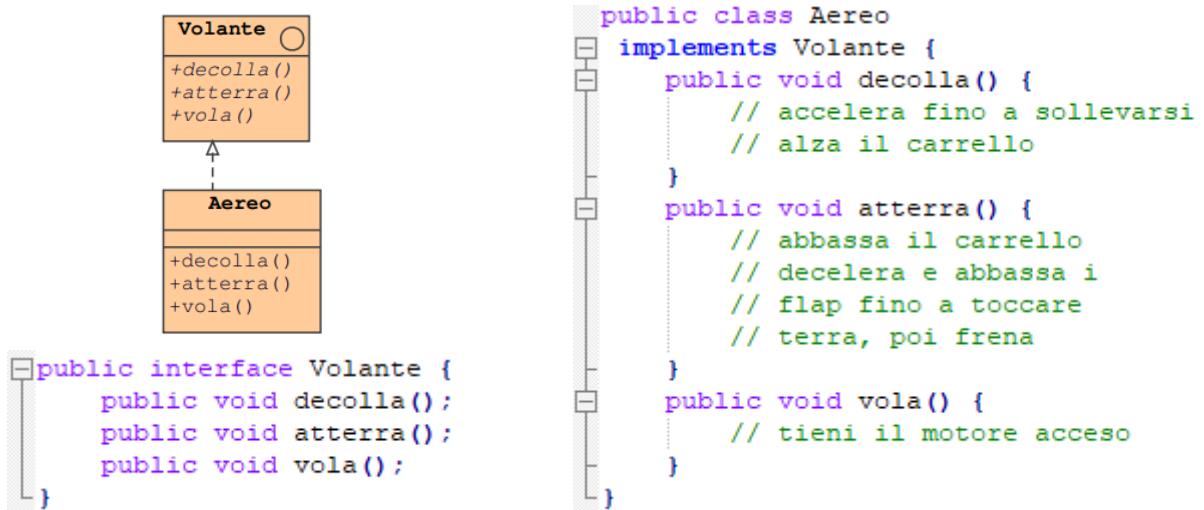
Nota: un interfaccia può anche dichiarare costanti public static final  
int COSTANTE = 7;

Attenzione: le variabili dichiarate nelle interfacce sono implicitamente *public static final*. La dichiarazione precedente poteva, quindi, anche essere scritta: int COSTANTE = 7;

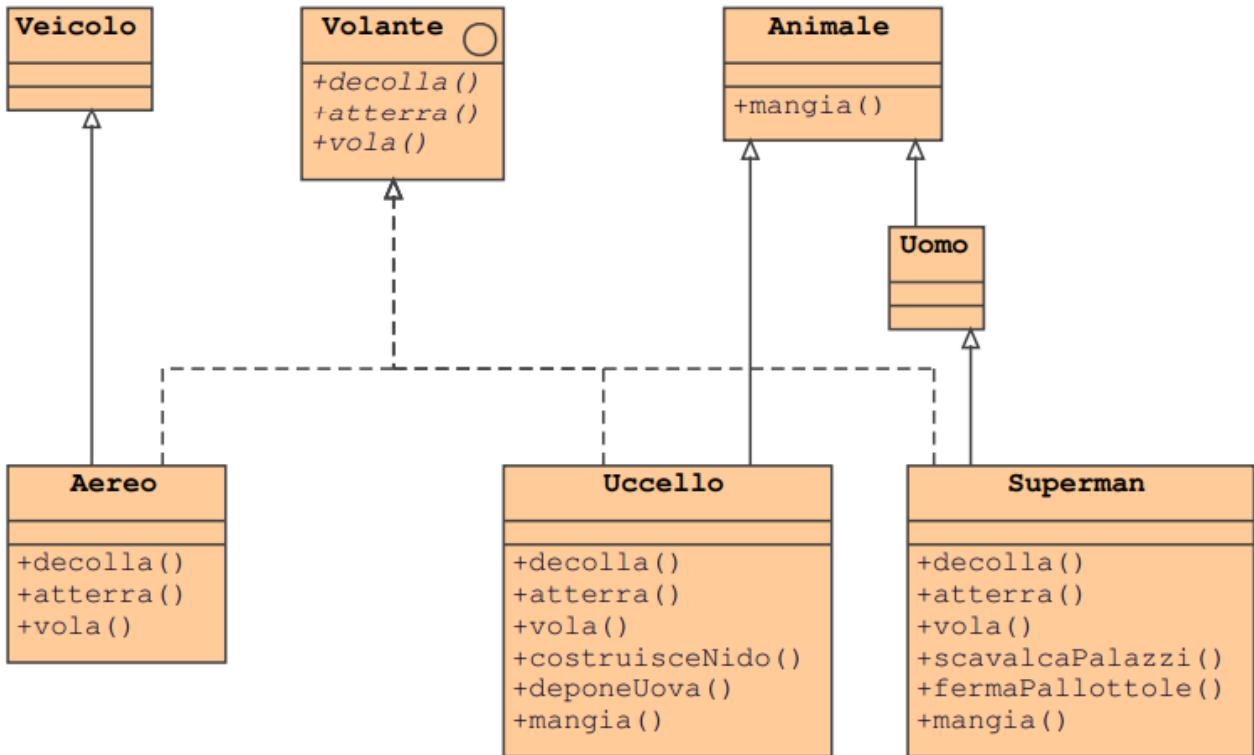
Attenzione: i metodi dichiarati nelle interfacce sono implicitamente *public abstract*.

### Esempio

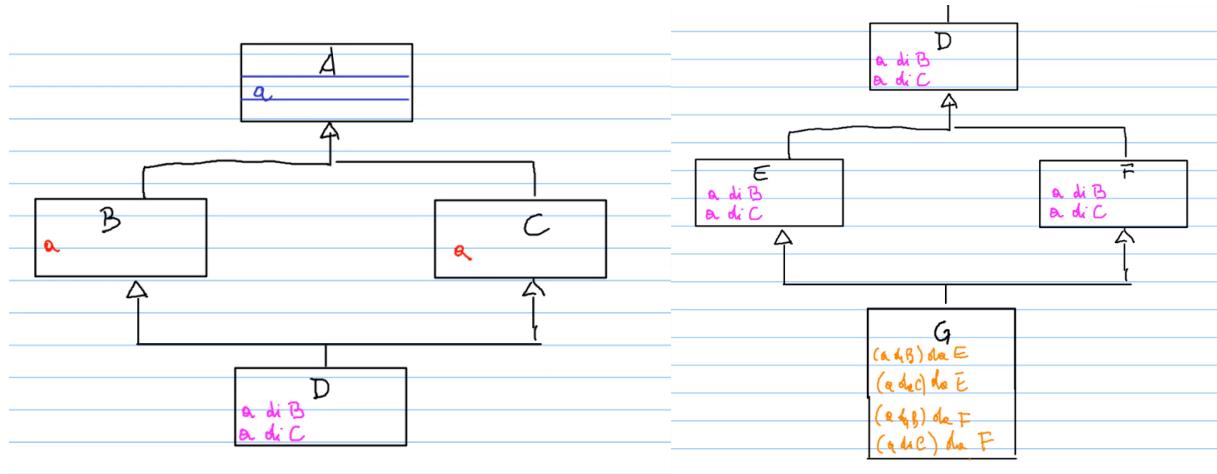
Immaginiamo un gruppo di oggetti che condividono la stessa abilità: essi volano. Si può costruire una interfaccia pubblica, chiamata Volante, che descriva tre operazioni: decolla, atterra e vola.



Volendo aggiungere altri oggetti volanti e definirli tramite una superclasse potremmo avere il seguente schema (Ma quali altri oggetti volano?, Qual è la superclasse di Uccello? Che cos'è un uccello?):

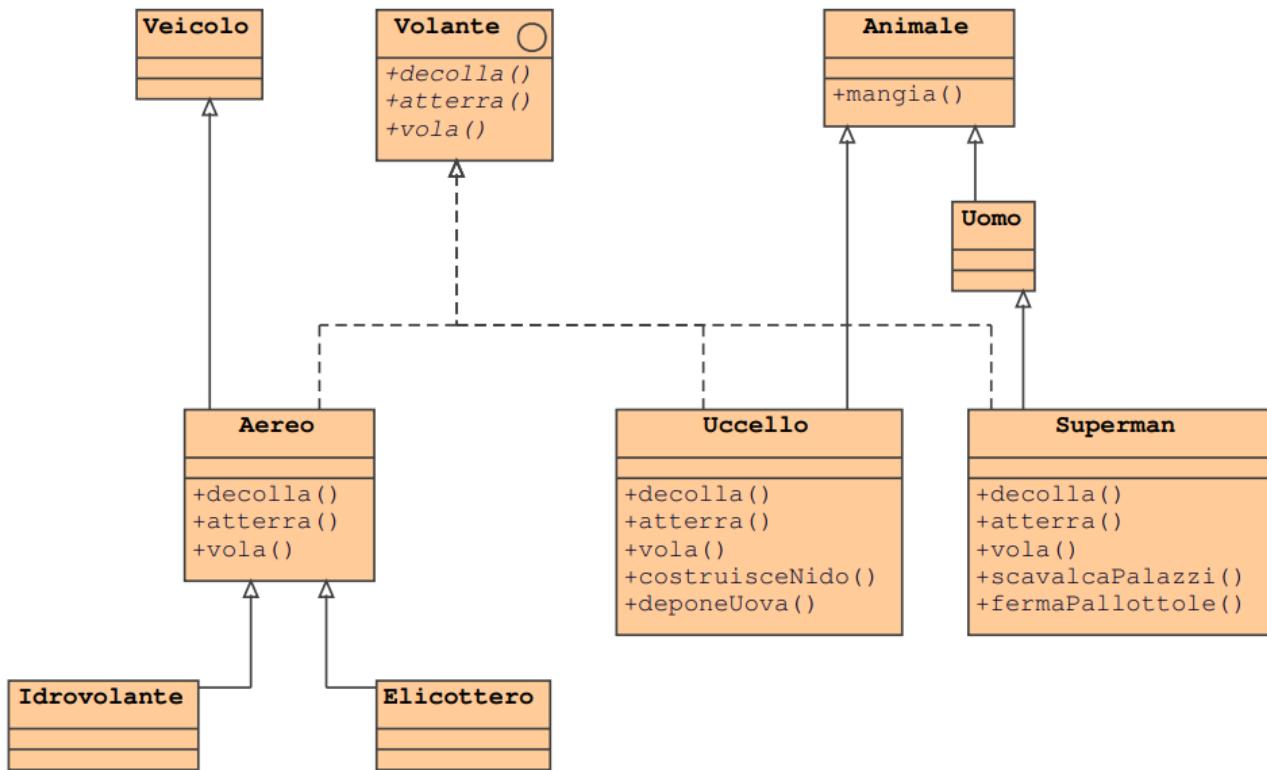


Anche se può sembrare un ereditarietà multipla grazie alle interfacce si evita il pericolo che una classe possa ereditare due distinte implementazioni dello stesso metodo. Per questo Java accetta ereditarietà multipla solo con al più una classe e una o più interfacce (che essendo astratte non può esserci alcuna ambiguità), a differenza del C++ che non ha questo tipo di limitazioni e quindi possono avvenire i seguenti errori: supponiamo di avere una gerarchia a diamante del seguente tipo:

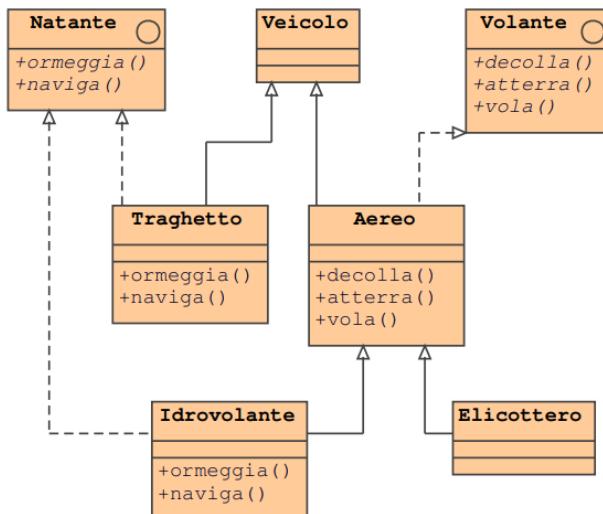


Dunque, avrei in D due copie della stessa variabile ma ereditata da percorsi diversi, mentre in G addirittura 4 copie della stessa variabile creando così delle ambiguità. Questo problema si risolve in C++ estendendo le classi in maniera virtuale così da far ereditare i metodi senza doppiarli (esempio: class B: public virtual A). Ma questo meccanismo viola il principio di object orientation di poter implementare una classe indipendentemente dalla sua ereditarietà.

Supponiamo adesso di voler costruire un sistema software di controllo dei voli. Esso dovrà garantire i permessi di decollo e di atterraggio a qualunque tipo di oggetto volante.



Una classe può implementare più di una interfaccia. Un *idrovoleante*, non solo vola, ma anche naviga. La classe *Idrovoleante* estende la classe *Aereo*, ed eredita l'implementazione dell'interfaccia *Volante*, ma implementa anche l'interfaccia *Natante*:



## Vantaggi delle interfacce

Le interfacce (il concetto di interfaccia è preso in prestito da *Objective-C*, in cui essa è chiamata *protocollo*) sono spesso considerate una alternativa all'ereditarietà multipla, anche se esse forniscono diversa funzionalità. Esse sono utili:

- per dichiarare metodi che una o più classi ci si aspetta dovrà implementare;
- per non rivelare il corpo vero di una classe, si rivela solo l'interfaccia (per esempio quando si distribuisce una classe ad altri sviluppatori di codice);
- per catturare similarità tra classi non correlate, senza forzare una relazione tra classi;
- per simulare l'ereditarietà multipla, dichiarando una classe che implementi varie interfacce.

## 46. Casting di riferimenti

### Introduzione

I riferimenti, come i primitivi, partecipano alla conversione automatica per assegnamento (e per passaggio di parametri) e al casting. Non c'è promozione aritmetica di riferimenti, poiché i riferimenti non possono essere operandi aritmetici, dunque, nella conversione e nel casting di riferimenti vi sono maggiori combinazioni tra vecchi e nuovi tipi (e maggiori combinazioni significano un numero maggiore di regole).

La conversione (automatica) di riferimenti avviene nella fase di compilazione, poiché il compilatore ha tutte le informazioni per determinare se la conversione è legale. Essa può essere forzata con un casting, come per i primitivi, ma ciò non significa che sia corretta (sarà poi java a controllare a run time che il casting sia corretto). In questo caso, infatti, può accadere che, sebbene la conversione sia autorizzata come legale dal compilatore, il tipo effettivo dell'oggetto riferito non sia compatibile in esecuzione (in una assegnazione entrano in gioco tre tipi: quello dei due riferimenti e quello proprio dell'oggetto).

### Conversioni automatiche

Le conversione automatiche sono le uniche che sicuramente funzionano e quindi il compilatore le accetta solo se non esiste nemmeno un caso possibile in cui la conversione non può avvenire.

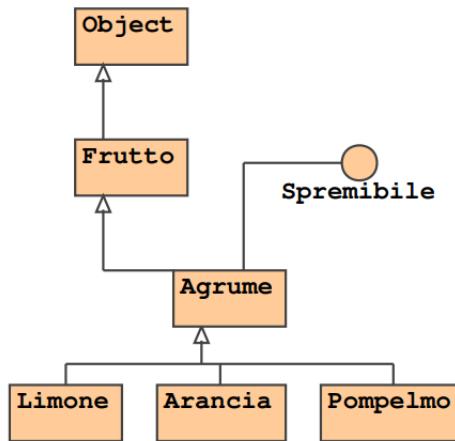
Avviene quando il tipo a destra dell'assegnamento è un sottotipo del tipo di sinistra, nella conversione automatica di riferimenti (per ampliamento, in una assegnazione o in un passaggio di parametri), vengono nascoste alcune caratteristiche dell'oggetto riferito. Qui il tipo di nascita dell'oggetto non interessa; i riferimenti possono essere ad una classe; ad una interfaccia e ad un array. La conversione in astratto può avvenire in: *Oldtype x = new Oldtype(); Newtype y = x;* (tra una variabile di vecchio tipo con una di nuovo tipo).

|                                  | <b>Oldtype è una classe</b>                     | <b>Oldtype è una interfaccia</b>                     | <b>Oldtype è un array</b>                                                                |
|----------------------------------|-------------------------------------------------|------------------------------------------------------|------------------------------------------------------------------------------------------|
| <b>Newtype è una classe</b>      | Oldtype deve essere sottoclasse di Newtype      | Newtype deve essere Object                           | Newtype deve essere Object                                                               |
| <b>Newtype è una interfaccia</b> | Oldtype deve implementare l'interfaccia Newtype | Oldtype deve essere una sotto-interfaccia di Newtype | Newtype deve essere Cloneable o Serializable                                             |
| <b>Newtype è un array</b>        | Errore di compilazione                          | Errore di compilazione                               | I tipi delle componenti dei due array devono essere <u>riferimenti</u> auto-convertibili |

Regola breve:

- Un tipo interfaccia può essere convertito solo ad un tipo interfaccia o ad Object. Se il nuovo tipo è una interfaccia essa deve essere una super-interfaccia di quella del vecchio tipo.
- Un tipo classe può essere convertito ad un tipo classe o ad un tipo interfaccia. Se convertito ad un tipo classe, il nuovo tipo deve essere una superclasse del vecchio tipo. Se convertito ad un tipo interfaccia, la vecchia classe deve implementare l'interfaccia.
- Un array può essere convertito alla classe Object, all'interfaccia Cloneable o all'interfaccia Serializable, oppure ad un array. Solo un array di riferimenti (non di primitivi) può essere convertito ad un array, e il vecchio tipo degli elementi deve essere convertibile al nuovo tipo degli elementi.

## Esempi



```

// L'upcast è sempre permesso e non da mai errore
Arancia arancia = new Arancia ();
Agrume agrume = arancia; // OK

// Il down cast non è detto che funzioni,
// di conseguenza occorrebbe un cast esplicito così
// da evitare l'errore in compilazione
// (ma potrebbe causare errore a runtime)
Agrume agrume = new Agrume ();
Arancia arancia = agrume; // NO

Pompelmo p = new Pompelmo ();
Spremibile s = p; // OK
Pompelmo p2 = s; // Errore di compilazione

```

**Regola:** un interfaccia può solo essere auto-convertita ad object. Ad esempio, nel seguente caso contano i tipi dei riferimenti negli elementi degli array:

```

Frutto frutta[];
Limone limoni[];
Agrume agrumi[] = new Agrume[10];
for (int i = 0; i < 10; i++) {
 agrumi[i] = new Agrume();
}
frutta = agrumi; // OK
limoni = agrumi; // Errore comp.

```

A proposito di overloading e overriding, in attinenza alle conversioni automatiche di riferimenti, ricordiamo che la scelta del metodo da associare ad una invocazione avviene in compilazione per i metodi sovraccaricati (basandosi sul tipo dei riferimenti nei parametri), ed in esecuzione per i metodi sovrapposti (basandosi sulla effettiva presenza e visibilità del metodo nell'oggetto).

```

class A { }
class B extends A {
 public void g(A a) {
 System.out.println("A");
 }
}
class UseAB extends B {
 public void f(A a) {
 System.out.println("A");
 }
 public void f(B a) {
 System.out.println("B");
 }
 public void g(A b) {
 System.out.println("B");
 }
 public static void main (String [] args) {
 UseAB u = new UseAB();
 u.metodo();
 }
}

public void metodo() {
 A a = new A();
 B b = new B();
 A x = b; // auto-conv.

 // esempio di overloading
 f(a); // stampa A
 f(b); // stampa B
 f(x); // stampa A !!!
 f(new B()); // stampa B

 // esempio di overriding
 g(a); // stampa B
 g(b); // stampa B
 g(x); // stampa B
 g(new B()); // stampa B
 super.g(a); // stampa A
 super.g(b); // stampa A
 super.g(x); // stampa A
}

```

## Casting esplicito

Una volta appurato quali sono le conversioni che il compilatore accetta di fare da sé (quelle per assegnazione, o passaggio di parametri, di ampliamento), possiamo passare a studiare le conversioni che il programmatore chiede esplicitamente (casting).

- Un cast di ampliamento, la cui conversione sarebbe avvenuta anche senza di esso, viene autorizzato e non causa problemi né in compilazione, né in esecuzione.
- Detto rozzamente, un casting di riferimenti verso l'alto, nella gerarchia di ereditarietà, è sempre, sia implicitamente sia esplicitamente, legale.
- Purtroppo, per i riferimenti e a differenza dei primitivi, nel casting verso il basso, il compilatore non può aiutare completamente e può accadere che, sebbene la conversione sia autorizzata come legale, essa fallisca in esecuzione.
- Infatti, nel casting esplicito, entrano in gioco tre tipi: quello dei due riferimenti (a sinistra e a destra dell'assegnazione) e quello vero (l'identità, l'imprinting di nascita) dell'oggetto.

Il casting esplicito avviene quando: Newtype nt; Oldtype ot; nt = (Newtype) ot ;

Regole per il compilatore:

|                                       | <b>Oldtype è una classe non-final</b>      | <b>Oldtype è una classe final</b>        | <b>Oldtype è una interfaccia</b>                      | <b>Oldtype è un array</b>                                                             |
|---------------------------------------|--------------------------------------------|------------------------------------------|-------------------------------------------------------|---------------------------------------------------------------------------------------|
| <b>Newtype è una classe non-final</b> | Oldtype deve estendere Newtype o viceversa | Oldtype deve estendere Newtype           | Sempre OK                                             | Newtype deve essere Object                                                            |
| <b>Newtype è una classe final</b>     | Newtype deve estendere Oldtype             | Oldtype e Newtype devono coincidere      | Newtype deve impl. l'interfaccia o impl. Serializable | Errore di compilazione                                                                |
| <b>Newtype è una interfaccia</b>      | Sempre OK                                  | Oldtype deve impl. l'interfaccia Newtype | Sempre OK                                             | Errore di compilazione                                                                |
| <b>Newtype è un array</b>             | Oldtype deve essere Object                 | Errore di compilazione                   | Errore di compilazione                                | I tipi delle componenti dei due array devono essere riferimenti convertibili per cast |

Regola breve (per il compilatore, nei casi più comuni):

- Quando sia Oldtype, sia Newtype sono classi, una classe deve essere sottoclasse dell'altra.
- Quando sia Oldtype, sia Newtype sono array, entrambi devono contenere riferimenti (non primitivi), e deve essere legale eseguire un cast tra gli elementi di Oldtype e quelli di Newtype.
- È sempre legale il cast tra una interfaccia e una classe non-final.

Infine, le ulteriori regole a cui deve obbedire un cast durante l'esecuzione (posto che sia sopravvissuto alla compilazione):

- Se Newtype è una classe, la classe originaria Oldtype dell'oggetto deve essere Newtype oppure una sua sottoclasse.
- Se Newtype è una interfaccia, la classe originaria Oldtype dell'oggetto deve implementare Newtype.

```

Limone l , ll;
Agrume a;
PompeLmo p;
l = new Limone ();
a = l; // auto-conver.
ll = (Limone) a; // cast legale
p = (PompeLmo) a; // cast illegale,
 // errore in esecuzione
/* Una interfaccia può essere
 auto-convertita solo ad Object. */
Spremibile s;
l = new Limone ();
s = l; // OK
ll = s; // errore di compilazione
l = (Limone) s ; // OK

```

```

/* E' sempre lecito il cast di una
 interfaccia ad una classe non-final
 in compilazione. Qui il cast ha
 successo anche in esecuzione. */

Limone l [];
Spremibile s [];
Agrume a [];
l = new Limone [7];
s = l; // OK
a = (Agrume []) s; // OK

```

## Questionario (fattili tu)

1) Quale dei seguenti enunciati è corretto?

- A. Solo i tipi primitivi sono convertiti automaticamente; per cambiare il tipo di un riferimento bisogna fare un cast.
- B. Solo i riferimenti sono convertiti automaticamente; per cambiare il tipo di un primitivo bisogna fare un cast.
- C. La promozione aritmetica di riferimenti richiede il cast esplicito.
- D. Sia i tipi primitivi, sia i riferimenti possono essere convertiti sia automaticamente sia attraverso un cast.
- E. Il cast dei tipi numerici richiede un controllo in esecuzione.

2) Quale dei seguenti enunciati è vero?

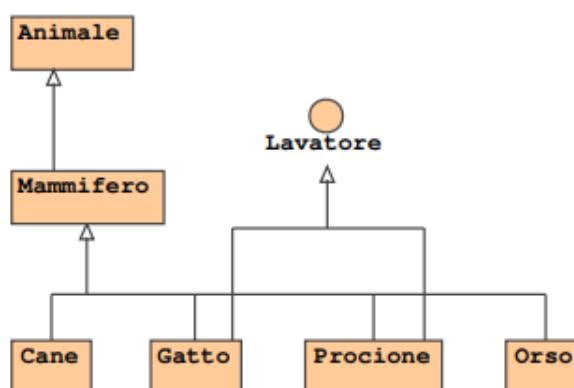
- A. I riferimenti ad oggetti possono essere convertiti nelle assegnazioni e non nelle invocazioni di metodi.
- B. I riferimenti ad oggetti possono essere convertiti nelle invocazioni di metodi e non nelle assegnazioni.
- C. I riferimenti ad oggetti possono essere convertiti sia nelle assegnazioni, sia nelle invocazioni di metodi, ma le regole nei due casi sono diverse.
- D. I riferimenti ad oggetti possono essere convertiti sia nelle assegnazioni, sia nelle invocazioni di metodi, e le regole nei due casi sono identiche.
- E. I riferimenti ad oggetti non possono essere mai convertiti.

3) Quale linea del codice seguente non compila?

```
1 Object ob = new Object();
2 String stringarr[] = new String [50];
3 Float floater = new Float(3.14f);
4
5 ob = stringarr;
6 ob = stringarr[5];
7 floater = ob;
8 ob = floater;
```

- A. La linea 5
- B. La linea 6
- C. La linea 7
- D. La linea 8

Per i prossimi tre quesiti si consideri la seguente gerarchia:



4) Si consideri il seguente codice:

```
1 Cane billy , fido;
2 Animale anim;
3
4 billy = new Cane();
5 anim = billy;
6 fido = (Cane) anim;
```

Quale dei seguenti enunciati è vero?

- A. La linea 5 non compila.
  - B. La linea 6 non compila.
  - C. Il codice è compilato correttamente ma viene lanciata una eccezione in esecuzione.
  - D. Il codice è compilato ed eseguito correttamente.
  - E. Il codice è compilato ed eseguito correttamente, ma il cast alla linea 6 è superfluo e può essere eliminato.
- 5) Si consideri il seguente codice. Quale dei seguenti enunciati è vero?
- ```
1 Gatto fufi;
2 Lavatore lala;
3 Orso bubu;
4
5 fufi = new Gatto();
6 lala = fufi ;
7 bubu = (Orso) lala;
```
- A. La linea 6 non compila poiché è richiesto un cast esplicito per convertire un Gatto ad un Lavatore.
 - B. La linea 7 non compila poiché non si può fare il cast da una interfaccia ad una classe.
 - C. Il codice è compilato ed eseguito correttamente ma il cast alla linea 7 non è necessario.
 - D. Il codice è compilato ma, alla linea 7, lancia una eccezione in esecuzione poiché la conversione a runtime di una interfaccia in una classe è proibita.
 - E. Il codice è compilato ma, alla linea 7, lancia una eccezione poiché il tipo di lala in esecuzione non può essere convertito al tipo Orso.
- 6) Si consideri il seguente codice. Quale dei seguenti enunciati è vero?
- ```
1 Procione rocco;
2 Orso bubu;
3 Lavatore lala;
4
5 rocco = new Procione();
6 lala = rocco;
7 bubu = lala;
```
- A. La linea 6 non compila: è richiesto un cast esplicito.
  - B. La linea 7 non compila: è richiesto un cast esplicito.
  - C. Il codice è compilato ed eseguito correttamente.
  - D. Il codice è compilato ma viene lanciata una eccezione alla linea 7, poiché in esecuzione la conversione da una interfaccia ad una classe è proibita.
  - E. Il codice è compilato ma viene lanciata una eccezione alla linea 7, poiché la classe dell'oggetto lala non può essere convertita al tipo Orso.

## 47. String e StringBuffer

### String

Le stringhe sono oggetti che possono essere creati come segue:

```
String s1 = new String ();
 s = "abcdef";

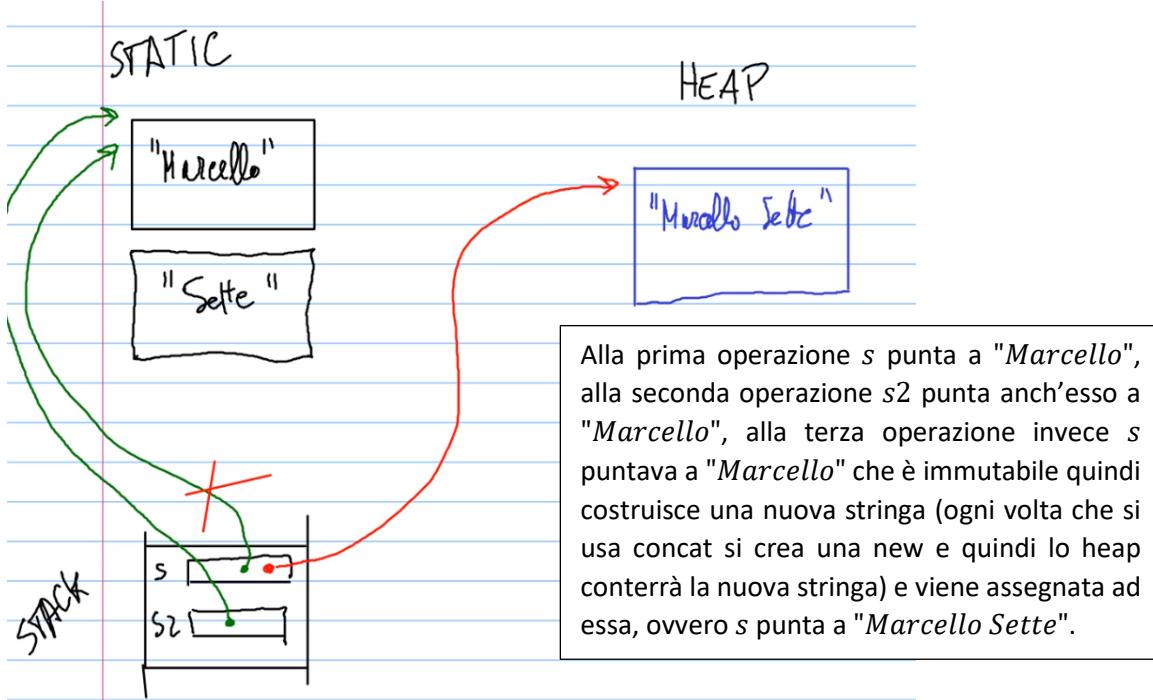
String s = new String ("abcdef");
 String s = "abcdef";
```

Ci sono delle sottili differenze tra i modi sopradescritti ma la cosa importante da capire è che le stringhe sono oggetti **immutabili**. Immutabilità significa che, una volta assegnato all'oggetto un valore, esso è fissato per

sempre (ed in seguito vedremo perché deve essere così). Attenzione: immutabili sono gli oggetti String (il contenuto), non i loro riferimenti. Questi ultimi, infatti, possono cambiare valore.

Di seguito alcuni esempi:

```
String s = "Marcello";
String s2 = s;
s = s.concat (" Sette"); // concatena la stringa s con " Sette"
System.out.println(s); // Stampa "Marcello Sette"
System.out.println(s2); // Stampa "Marcello"
```



```
String x = "Marcello";
x.concat(" Sette");
System.out.println(x); // Stampa Marcello
```

Anche l'operatore + funziona come la concat:

```
// Quanti oggetti in gioco?
String s1 = "A";
String s2 = s1 + "B";
s1.concat("C");
s2.concat(s1);
s1 += "D";
System.out.println(s1 + s2); // Stampa ADAB
```

Uno alla prima riga, alla seconda si crea un oggetto `s2` e uno "B", alla terza una stringa grazie alla concat ed un "C", alla quarta si viene a creare una stringa dalla concat, alla quinta ancora una concatenazione ed un "D", anche la print line mi da un nuovo oggetto, quindi in tutto sono nove oggetti.

```
String s1 = " abc ";
String s2 = s1 + ""; // crea praticamente una copia di s1
String s3 = " abc ";
System.out.println(s1 == s2); // false!
System.out.println(s1 == s3); // true!
```

La prima print line stamperà falso poiché si ricorda che l'operatore == applicato a string confronta i puntatori. Mentre la seconda stampa true poiché se si ha già un letterale stringa uguale la JVM non costruisce

un nuovo oggetto ma fa puntare quest'ultimo allo stesso oggetto (funziona solo per le stringhe definite con apici), questo è possibile proprio grazie alla proprietà di immutabilità delle stringhe.

Per motivi di efficienza, poiché nelle applicazioni i littorali String occupano molta memoria, la JVM riserva un'area speciale di memoria ad essi: la **String constant pool**. Quando il compilatore incontra un littorale String, esso controlla che non sia già presente nel pool. Se è presente, allora il riferimento al nuovo littorale è diretto alla stringa esistente (la stringa esistente ha semplicemente un ulteriore riferimento), altrimenti lo aggiunge al pool. Se ci sono molti riferimenti, in punti diversi del codice, allo stesso littorale, sarebbe deleterio permettere la modifica del littorale usando uno solo di tali riferimenti. Quindi ora non solo si capisce perché gli oggetti String devono essere immutabili, ma anche la differenza tra i due seguenti enunciati:

```
String s = "abcdef"; // non costruisce sempre un nuovo oggetto

String s = new String("abcdef"); // crea sempre un nuovo oggetto
```

## StringBuffer

Se proprio si deve fare un uso intensivo di manipolazione di stringhe, allora è opportuno usare gli oggetti StringBuffer: essi sono un po' come gli oggetti String, ma non sono immutabili. Per esempio:

```
StringBuffer s = new StringBuffer("Marcello");
s.append(" Sette"); // modifica l'oggetto s
System.out.println(s); // stampa "Marcello Sette"
```

Attenzione: mentre la classe String sovrappone il metodo equals in modo da controllare l'uguaglianza del contenuto dei due oggetti (quello corrente e quello ricevuto come parametro), la classe StringBuffer non lo sovrappone ed usa quello ereditato da Object che funziona essenzialmente come l'operatore == (cioè compara i riferimenti). StringBuffer e string non sono auto-convertibili quindi scrivere StringBuffer s = "abc"; è illegale e non è neppure possibile convertire string e StringBuffer tramite cast esplicativi, ovvero anche StringBuffer s = (StringBuffer) "abc" è illegale (ci sono degli opportuni metodi che però lo consentono).

Le classi String e StringBuffer sono final. Esse non possono essere specializzate in modo da sovrapporre i loro metodi: l'invocazione virtuale di metodi sarebbe un grave problema di sicurezza.

## 48. Garbage collector

### Funzionamento

Riprendiamo il funzionamento della memoria: abbiamo una zona statica dove ho le classi che comprendono lo string pool (dove finiscono le stringhe tra apici) e attributi statici, abbiamo poi lo stack di attivazione dove al solito ci sono parametri, variabili locali e informazioni aggiuntive come i puntatori di ritorno; infine c'è lo heap con oggetti e istanze delle varie classi (quasi tutte, infatti con lo string pool alcune stringhe finiscono nella zona statica).

Un oggetto è in uso quando c'è un modo di riferirsi a quell'oggetto; ci si può riferire ad un oggetto con una variabile o con un attributo (comprende i puntatori). Quindi per vedere se un oggetto è in uso si può far riferimento alle variabili locali nello stack o agli attributi statici nello static, dove entrambi punteranno a oggetti dello heap. Variabili e attributi possono cambiare durante l'esecuzione del programma e quindi liberare oggetti nello heap.

Il garbage collector dunque guarda tutti i tipi reference nello stack e nello heap marcando tutti gli oggetti raggiungibili, e va a deallocate tutti gli oggetti che non sono stati precedentemente marcati (identificandoli e rimuovendoli). Ci sono vari algoritmi di come possa funzionare il garbage collector ma tutti partono dall'idea di poter deallocate tutti gli oggetti liberi, ovviamente non si sa quando effettivamente gli oggetti vengono

deallocati ma si può dire per certo quando diventano eleggibili per il garbage collector (quando vengono liberati).

## Esempi di eleggibilità

```

public class Esempio1 {
 public static void main (String [] args) {
 StringBuffer sb = new StringBuffer ("Ciao");
 System.out.println (sb);
 // l'oggetto riferito da sb non è ancora eleggibile per GC
 sb = null; // ora è eleggibile.
 }
}

public class Esempio2 {
 public static void main (String [] args) {
 StringBuffer s1 = new StringBuffer ("Ciao");
 StringBuffer s2 = new StringBuffer ("Addio");
 System.out.println(s1);
 // l'oggetto riferito da s1 non è ancora eleggibile per GC
 s1 = s2; // ora è eleggibile.
 }
}

import java.util.Date;
public class Esempio3 {
 public static void main (String [] args) {
 Date d = getDate();
 System.out.println(d);
 } // alla fine del main viene liberato d2
 public static Date getDate() {
 Date d2 = new Date();
 String now = d2 . toString ();
 System.out.println(now);
 return d2;
 } // alla fine di getDate viene liberato now
}

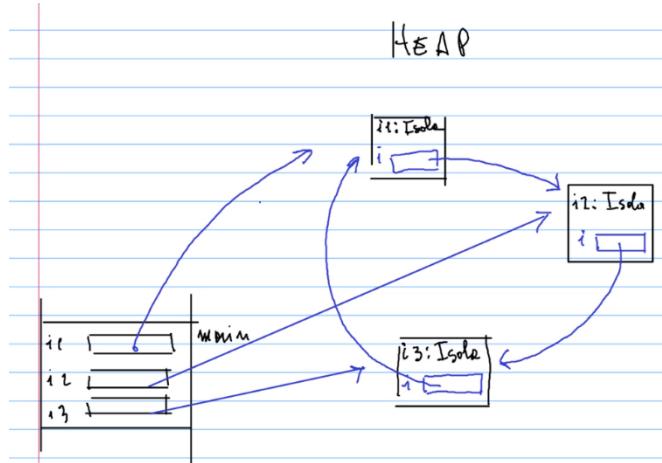
public class Isola {
 Isola i;

 public static void main (String [] args) {
 Isola i1 = new Isola();
 Isola i2 = new Isola();
 Isola i3 = new Isola();

 i1.i = i2;
 i2.i = i3;
 i3.i = i1;

 i1 = null;
 i2 = null;
 i3 = null;
 // solo qui vengono resi
 // eleggibili tutti e
 // tre i riferimenti
 }
}

```



Esempio importante poiché fa capire come mai il GB deve effettivamente marcare gli oggetti prima di poterli deallocare, infatti una possibile idea di GC potrebbe essere mantenere in ogni oggetto un contatore che dice quanti puntatori puntano a quell'oggetto, nello schema precedente avremo quindi un conteggio di 2 per ogni

istanza di *i*, di conseguenza quando elimino il puntatore *i1* decremento il contatore di *i1*, quando elimino *i2* decremento il contatore di *i2*, quando elimino *i3* decremento *i3*, l'idea è praticamente di deallocare l'oggetto quando il contatore va a 0, ma il problema è che quando la struttura è ciclica come nell'esempio i 3 oggetti nello heap non sono più raggiungibili ma si mantengono a vicenda, dunque questo algoritmo non funziona bene e crea un fenomeno chiamato memory leakage.

## 49. Eccezioni (Gestione degli errori)

### try, catch e (finally)

Le eccezioni denotano "eventi eccezionali" la cui occorrenza altera il flusso normale delle istruzioni, ad esempio risorse hardware indisponibili (ad es. la rete), hardware malfunzionante, bachi nel software, etc...

Quando capita un tale evento, si dice che viene "lanciata una eccezione". Per gestire le eccezioni bisogna inglobare il codice che potrebbe lanciare una eccezione in un blocco marcato try. Il codice che assume la responsabilità di fare qualcosa in conseguenza del lancio di una eccezione si chiama manipolatore (exception handler) e va inglobato in una clausola catch. Inoltre, è possibile aggiungere un blocco opzionale marcato finally che verrà sempre eseguito, anche dopo il lancio e la eventuale manipolazione dell'eccezione:

```
try {
 // Qui va scritto il codice "rischioso"
}
catch (Eccezione1 e) {
 // Qui il codice che manipola una Eccezione1
}
catch (Eccezione2 e) {
 // Qui il codice che manipola una Eccezione2
}
finally {
 // Codice da eseguire in ogni caso
}
// codice non rischioso va scritto qui
```

Il blocco finally viene eseguito perfino dopo una eventuale istruzione return presente nei blocchi try o catch ma potrebbe non essere eseguito o potrebbe non completare l'esecuzione solo in conseguenza di un crash totale del sistema oppure tramite una invocazione di *System.exit(int status)* (interrompe la JVM).

### Vincoli

Le clausole catch ed il blocco finally sono opzionali ma dopo un blocco try deve esistere almeno una clausola catch poiché un blocco try solitario causa un errore di compilazione.

Se esistono una o più clausole catch esse devono seguire immediatamente il blocco try, mentre se esiste il blocco finally esso deve seguire l'ultima clausola catch. Si noti che non è ammessa nessuna istruzione tra il blocco try, le clausole catch ed il blocco finally. Significativo, inoltre, è l'ordine in cui si succedono tra loro le clausole catch (vedremo in seguito).

### Propagazione

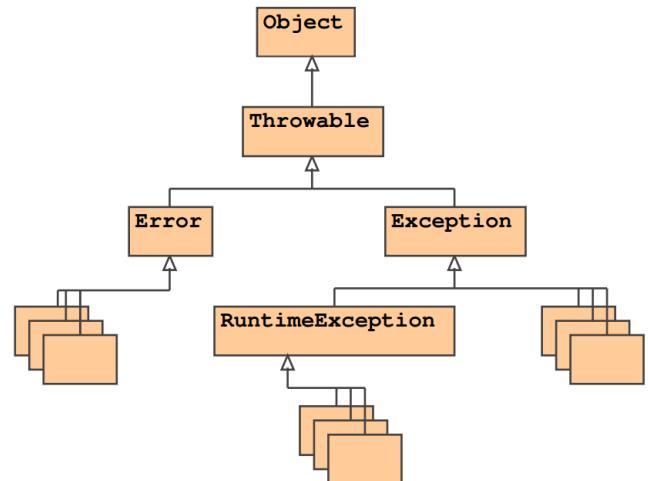
Le clausole catch non sono obbligatorie poiché c'è la possibilità che l'errore sia irrecuperabile e si lascia che il programma termini. Di contro, una eccezione che viene lanciata da un blocco try, ma per la quale non esiste una clausola catch che l'attende si dice non catturata. Una eccezione non catturata semplicemente si "si immerge" nel record di attivazione che la ha generata, "riemergendo" nel successivo record di attivazione dove potrebbe essere catturata da una ulteriore clausola catch, oppure altrimenti ricadere nel record successivo, e così via. Finché, eventualmente, l'eccezione raggiunge il record di attivazione del main, dal quale, se non catturata "esplode" (il programma termina generando un errore), producendo, se possibile, una descrizione del proprio percorso (stack trace).

## Definizione

In Java tutto ciò che non è primitivo è un oggetto, e le eccezioni non fanno “eccezione” a questa regola, difatti ogni eccezione è una istanza di una sottoclasse della classe di Exception al fine di poter distinguere i vari tipi di eccezioni che possono essere lanciate in un programma.

Una eccezione viene lanciata usando la parola riservata throw: `throw new Exception();` (è solitamente seguita dalla new). Tutto ciò che segue, nello stesso blocco di istruzioni, il lancio dell’eccezione non verrà eseguito (tranne l’eventuale blocco finally). La gerarchia degli oggetti è la seguente:

Gli oggetti che si possono lanciare con l’Exception sono una sottoclasse dell’oggetto Throwable (superclasse anche dell’oggetto Error). La classe Throwable ha il metodo `printStackTrace()` (questo metodo aiuta il programmatore a capire dove si è verificato l’errore stampando il numero di riga in cui si è verificata l’eccezione). La classe Error e le sue sottoclassi rappresentano situazioni insolite che non sono causate da errori di programmazione o da ciò che normalmente succede durante l’esecuzione del programma. Per esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa non è disponibile. Generalmente una applicazione non deve essere capace di riprendersi da una situazione di errore, pertanto un programma non è obbligato a gestire gli oggetti Error: esso compila senza problemi.



Un ruolo simile è rappresentato dalla classe RuntimeException, questa rappresenta pure eventi eccezionali ma dovuti al programma (errori di programmazione, bachi), inoltre, indicano eccezioni rare e difficili da gestire. Il programmatore, che si accorge di un baco dovuto ad un suo errore, deve correggerlo, non gestirlo come una eccezione!

## Eccezioni catturate

Una clausola catch cattura ogni oggetto-eccezione il cui tipo può essere ricondotto mediante conversione automatica al tipo specificato nella clausola (il tipo è esattamente quello della clausola catch oppure tutto quello che può essere convertito nella clausola catch). Esempio: la classe `IndexOutOfBoundsException` ha due sottoclassi, `ArrayIndexOutOfBoundsException` e `StringIndexOutOfBoundsException`; si può scrivere una unica clausola che catturi una qualunque di queste eccezioni:

```
try {
 // Codice che potrebbe lanciare una eccezione
 // IndexOutOfBoundsException oppure
 // ArrayIndexOutOfBoundsException oppure
 // StringIndexOutOfBoundsException
}
catch (IndexOutOfBoundsException e) {
 e.printStackTrace();
}
```

Bisogna resistere alla tentazione di scrivere una unica clausola catch-all poiché avere catch diversificati aiuta a identificare meglio il problema allo stesso programmatore. Inoltre, l’ordine nelle clausole catch è, come accennato precedentemente, importante. Se, all’esempio precedente, avessimo scritto:

```

try {
 // Codice che potrebbe lanciare una eccezione
 // IndexOutOfBoundsException oppure
 // ArrayIndexOutOfBoundsException
}
catch (IndexOutOfBoundsException e) {
 // tratta l'eccezione
}
catch (ArrayIndexOutOfBoundsException e) {
 // tratta l'eccezione
}

```

Il codice non sarebbe compilato poiché i catch vengono eseguiti nell'ordine e quindi verrebbe eseguito sempre e solo il primo catch e il compilatore lancia un errore. Vanno messi prima gli eventuali catch prima della superclasse (spesso è domanda di esame capire se l'ordine delle eccezioni è errato)

Così come la dichiarazione del metodo deve specificare il numero e il tipo dei parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare devono essere dichiarate (a meno che non siano sottoclassi di RuntimeException). La parola chiave throws viene usata per elencare le eccezioni che possono fuoriuscire da un metodo: `void miaFunzione() throws MiaEccezione1, MiaEccezione2{ codice per il metodo }`; il fatto che un metodo dichiara l'eccezione non significa che esso la lancerà sempre, ma avverte l'utilizzatore che esso potrebbe lanciarla.

## Handle or Declare

Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola handle or declare, volendo si possono usare entrambe):

- 1) Gestire l'eccezione fornendo le opportune sezioni try/catch.
- 2) Propagare l'eccezione dichiarandola nell'intestazione del metodo

Le RuntimeException sono esenti dall'obbligo di dichiarazione (fanno da "eccezione" alla regola precedente). Esse sono unchecked (non controllate) dal compilatore, mentre le rimanenti eccezioni sono dette checked.

## Esempi

Quali problemi ci sono in questo codice?

```

import java.io.*;
class Test {
 public int f1() throws EOFException {
 return f2();
 }
 public int f2() throws EOFException {
 // qui il codice che lancia effettivamente l'eccezione
 return 1;
 }
}

```

Poiché `EOFException` è sottoclasse di `IOException`, che è sottoclasse di `Exception`, essa è una eccezione checked e come tale viene regolarmente dichiarata e quindi il codice regolarmente compilato.

Ed in questo codice?

```

public void f1 () {
 // qui codice che può lanciare NullPointerException
}

```

Poiché `NullPointerException` è sottoclasse di `RuntimeException`, essa è una eccezione unchecked. Non è necessario né dichiararla, né catturarla, ed il codice viene regolarmente compilato.

## Nuove eccezioni

È possibile usare tipi di eccezioni già presenti nelle Java API, oppure crearne di propri in questo modo:  
`class MiaEccezione extends Exception { }`, oppure estendendo una qualunque sottoclasse di `Exception`. Da questo momento in poi si può lanciare un oggetto del tipo (checked) `MiaEccezione`.

## Overriding

Essendo l'overriding una delle caratteristiche dei metodi anche qui è consentita la sovrapposizione dei metodi. Posto, però, che sono sovrapponibili solo i metodi visibili della superclasse, di seguito si riportano le regole complete per la sovrapposizione di metodi:

- I due metodi devono avere identica segnatura.
- I due metodi devono avere identico tipo di ritorno.
- Non si può marcare static uno solo dei metodi (o nessuno o entrambi).
- Il metodo nella superclasse non può essere marcato final.
- Il metodo nella sottoclasse deve avere visibilità non inferiore a quello della superclasse (la visibilità può solo crescere).
- Il metodo nella sottoclasse può dichiarare di lanciare un tipo di eccezione checked, ma tale tipo non deve essere un nuovo tipo o un tipo più “esteso” rispetto a quelli dichiarati dal metodo della superclasse. Cioè, le eventuali eccezioni checked dichiarate dal metodo nella sottoclasse, devono essere tipi posti al di sotto nella gerarchia delle eccezioni dichiarate dal metodo nella superclasse (non si devono introdurre nuovi tipi di eccezioni non previste nella superclasse).

## Questionario (soluzioni in fondo)

1) Dato il codice a destra. Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccB?

- A. 1      B. 2      C. 3  
D. 4      E. 5      F. 6

```
try {
 // codice rischioso; hp:
 // Exception
 // +--- EccA
 // +--- EccB
 // +--- EccC
 System.out.print(1);
}
catch (EccB e) {
 System.out.println(2);
}
catch (EccC e) {
 System.out.println(3);
}
catch (Exception e) {
 System.out.println(4);
}
finally {
 System.out.println(5);
}
System.out.println(6);
```

2) Dato il codice a destra. Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 non venga lanciata alcuna eccezione?

- A. 1      B. 2      C. 3  
D. 4      E. 5      F. 6

3) Dato il codice a destra. Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccC?

- A. 1      B. 2      C. 3  
D. 4      E. 5      F. 6

4) Dato il codice a destra. Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccA?

- A. 1      B. 2      C. 3      D. 4      E. 5      F. 6

5) Dato il codice in altro a destra. Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo Exception?

- A. 1      B. 2      C. 3      D. 4      E. 5      F. 6

6) Dato il codice in alto a destra. Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo RuntimeException?

- A. 1      B. 2      C. 3      D. 4      E. 5      F. 6

**7)** Dato il codice alla pagina precedente. Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo Error?

- A. 1      B. 2      C. 3      D. 4      E. 5      F. 6

**8)** Qual è l'output del programma precedente?

- A. 5      B. 14      C. 124      D. 156      E. 1245      F. 35

**9)** Quali affermazioni riguardanti il seguente programma sono vere?

```
public class Eccezioni {
 public static void main (String [] args) {
 try {
 if (args.length == 0) return;
 System.out.println(args[0]);
 }
 finally {
 System.out.println("Fine");
 }
 }
}
```

- A. Se eseguito senza argomenti, il programma non produce output.
- B. Se eseguito senza argomenti, il programma stampa Fine.
- C. Il programma lancia un ArrayIndexOutOfBoundsException.
- D. Se eseguito con un argomento, il programma stampa solo l'argomento dato.
- E. Se eseguito con un argomento, il programma stampa l'argomento dato seguito da Fine.

**10)** Qual è l'output del seguente programma?

```
public class MyClass {
 public static void main (String [] args) {
 RuntimeException re = null;
 throw re;
 }
}
```

- A. Il codice non viene compilato, poiché il main non dichiara che lancia una RuntimeException.
- B. Il codice non viene compilato, poiché non può rilanciare *re*.
- C. Il programma viene compilato e lancia java.lang.RuntimeException in esecuzione.
- D. Il programma viene compilato e lancia java.lang.NullPointerException in esecuzione.
- E. Il programma viene compilato, eseguito e termina senza produrre alcun output.

**11)** Quali di queste affermazioni sono vere?

- A. Se una eccezione non è catturata in un metodo, il metodo termina e viene ripresa la successiva normale esecuzione.
- B. Un metodo sovrapposto in una sottoclasse deve dichiarare che lancia lo stesso tipo di eccezione del metodo che sovrappone.
- C. Il main può dichiarare che lancia eccezioni checked.
- D. Un metodo che dichiara di lanciare una certo tipo di eccezione, può lanciare una istanza di una qualunque sottoclasse di quel tipo.
- E. Il blocco finally è eseguito se e solo se viene lanciata una eccezione all'interno del corrispondente blocco try.

12) Qual è l'output del programma seguente?

```
public class MiaClasse {
 public static void main (String [] args) {
 try {
 f();
 }
 catch (MiaEcc e) {
 System.out.print (1);
 throw new RuntimeException();
 }
 catch (RuntimeException e) {
 System.out.print(2);
 return;
 }
 catch (Exception e) {
 System.out.print (3);
 }
 finally {
 System.out.print(4);
 }
 System.out.print(5);
 }
 // MiaEcc è sottoclasse di Exception
 static void f() throws MiaEcc {
 throw new MiaEcc();
 }
}
```

- A. 5      B. 14      C. 124      D. 145      E. 1245      F. 35

13) Qual è l'output del programma seguente?

```
public class MiaClasse {
 public static void main (String [] args) throws MiaEcc {
 try {
 f();
 System.out.print(1);
 }
 finally {
 System.out.print(2);
 }
 System.out.print(3);
 }
 // MiaEcc è sottoclasse di Exception
 static void f() throws MiaEcc {
 throw new MiaEcc();
 }
}
```

- A. 2 e lancia MiaEcc    B. 12    C. 123    D. 23    E. 32    F. 13

(1) B E e F; Viene catturata l'eccezione, eseguito il finally ed infine l'esecuzione continua normalmente blocco finally, l'eccezione propagata. (8) D (9) B e (10) D (11) C e D (12) B (13) A  
(2) A E e F; Viene compilato il blocco try, eseguito il finally, infine l'esecuzione continua normalmente (3) C E e F (4) D E e F (5) D, E e F (6) D, E e F (7) E; non viene catturata l'eccezione, viene eseguito il blocco finally, l'eccezione propagata.

## 50. Polimorfismo parametrico vs polimorfismo per inclusione

### Esercizio

Definire il tipo di dato "Stack" con operazioni Push(element) e Pop() senza "forzare" una specifica implementazione o un tipo specifico per gli elementi. Definire poi l'implementazione a lista concatenata: LinkedStack.

### Una soluzione

```
public interface ObjectStack {
 public void push(Object el);
 public Object pop() throws EmptyStackException;
}

public class EmptyStackException extends Exception {
 static final long serialVersionUID = 99999;
}

public class LinkedObjectStack implements ObjectStack {
 private class StackRecord {
 Object el;
 StackRecord next;
 StackRecord(Object el, StackRecord next) {
 this.el = el;
 this.next = next;
 }
 }

 StackRecord top;
 public void push(Object el) {
 top = new StackRecord(el, top);
 }
 public Object pop() throws EmptyStackException {
 Object res;
 if(top == null) {
 throw new EmptyStackException();
 }
 res = top.el;
 top = top.next;
 return res;
 }
}
```

### Esempio di uso:

```
ObjectStack so = new LinkedObjectStack();

so.push(1); // conversione implicita a Integer (autoboxing)
so.push(2); // e upcast automatico a Object
so.push(3);

try{
 while(true){
 System.out.println(((Integer)so.pop()).intValue());
 }
} catch(Exception e) { System.out.println("fine"); }
```

Se non faccio il downcast...

non posso usare questo metodo  
(specifico di Integer)

```
3
2
1
fine
```

output

```

ObjectStack so = new LinkedObjectStack();

so.push(1); // conversione implicita a Integer (autoboxing)
so.push(2); // e upcast automatico a Object
so.push("A");
try{
 while(true){
 System.out.println(((Integer)so.pop()).intValue());
 }
} catch(Exception e) { System.out.println("fine"); }

```

Se mi distraggo compila ancora ma...

output

```
java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
```

### Una soluzione migliore: template

```

public interface GenericStack<ElemType> {
 public void push(ElemType el);
 public ElemType pop() throws EmptyStackException;
}

public class GenericLinkedStack<ElemType> implements GenericStack<ElemType> {
private class StackRecord<ElType> {
 ElType el;
 StackRecord ElType next;
 StackRecord(ElType el, StackRecord ElType next) {
 this.el = el;
 this.next = next;
 }
}
StackRecord ElemType top;
public void push(ElemType el) {
 top = new StackRecord ElemType(el, top);
}
public ElemType pop() throws EmptyStackException {
 ElemType res;
 if(top == null) {
 throw new EmptyStackException();
 }
 res = top.el;
 top = top.next;
 return res;
}
}

```

Nel momento in cui vado ad usare il template devo sostituire il parametro attuale con `ElemType`, in questo modo non è più necessario il cast poiché so che il risultato della `pop()` è già `integer`; di conseguenza venendo meno il `downcast` vengono meno anche i suoi possibili errori, infatti prendendo la stessa situazione precedente non avremo un errore a run time ma dallo stesso compilatore. Esempio d'uso:

```

GenericStack< Integer > si = new GenericLinkedStack< Integer >();

si.push(1);
si.push(2);
si.push(3);
try{
 while(true){ System.out.println(si.pop().intValue()); }
}
catch(Exception e) { System.out.println("fine"); }

```

Non serve downcast:  
pop() restituisce Integer

```

3
2
1
fine

```

output

```

GenericStack< Integer > si = new GenericLinkedStack< Integer >();

si.push(1);
si.push(2);
si.push("A");
try{
 while(true){ System.out.println(si.pop().intValue()); }
}
catch(Exception e) { System.out.println("fine"); }

```

push( Integer )

1. ERROR in mylists/Test2.java (at line 8)

```

si.push("A");
^^^^

```

output del compilatore!

The method push(Integer) in the type GenericStack<Integer> is not applicable  
for the arguments (String)

I template prendono il nome di polimorfismo parametrico, questi rendono il sistema più robusto e permette al compilatore di fare tutti i controlli necessari e non richiede controlli run time infatti gli errori vengono anticipati a tempo di compilazione.

## Confronto

I vantaggi del polimorfismo parametrico è l'anticipo della scoperta di errori di tipo a tempo di compilazione evitando i controlli di tipo a run time, mentre il polimorfismo per inclusione permette strutture dati eterogenee, ad esempio, Stack di elementi di tipo diverso. In molti casi ci servono collezioni di elementi eterogenei ma vogliamo comunque usarli allo stesso modo, praticamente vogliamo prendere il meglio da entrambi i polimorfismi.

Ad esempio, un possibile scena è una *lista* di forme eterogenee (rettangoli, ellissi, linee, ecc.) in cui vogliamo semplicemente usare la lista per implementare *refresh*, che deve solo inviare un messaggio *draw()* a tutte le forme della lista, quindi bisogna identificare le modalità d'uso degli elementi; scegliere una superclasse comune o fattorizzarle in una interfaccia e, infine, usare la superclasse/interfaccia come argomento del template.

## Note sull'implementazione

In Java - internamente - sarebbe comunque uno stack di Object

- Ma coi template dichiariamo che vogliamo metterci solo oggetti di un certo tipo
- Per questo non possiamo legare il parametro a tipi elementari come int o float ma dobbiamo usare i wrapper

In C++ ogni uso dei template fa compilare una nuova classe

- Sorta di macro
- Il compilatore inserisce nel programma la definizione di classe specializzata e la ricompila per ogni parametro attuale

## 51. Classi interne

### Introduzione

Come si è notato nel capitolo precedente (ove nella seconda parte della prima soluzione si implementa la classe interna *LinkedObjectStack*) è utile avere una classe interna ad un'altra classe poiché questa può essere usata (se dichiarata private) solo dalla classe in cui è dichiarata. Un esempio concreto di quanto siano utili le interfacce si ha con l'implementazione di una GUI (Graphical User Interface), ne portiamo di seguito un primo esempio:

```
import javax.swing.*;

public class ProvaGUI {
 private JFrame frame;
 private JButton pulsante;

 public static void main (String [] arg) {
 ProvaGUI gui = new ProvaGUI();
 gui.go ();
 }
 public void go() {
 frame = new JFrame();
 pulsante = new JButton ("Cliccami");

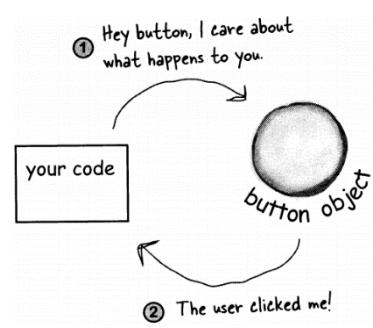
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frame.getContentPane().add(pulsante);
 frame.setSize(300, 300);
 frame.setVisible(true);
 }
}
```

Ma non c'è alcuna azione collegata alla pressione del pulsante.

### Gestione degli eventi

Piuttosto che controllare ciclicamente e continuamente lo stato dell'oggetto pulsante, è molto più efficiente lasciare che sia l'oggetto pulsante a comunicarci il proprio cambiamento di stato, rendendo il nostro codice un *ascoltatore* e il pulsante è la *sorgente*.

La comunicazione tra sorgente e ascoltatore (è una interfaccia) avviene con l'utilizzo di metodi particolari, uno nella quale si registra il pulsante passando this come parametro attuale, così da dare modo al destinatario di richiamare la classe ascoltatore. Il secondo metodo è quello chiamato dalla classe sorgente che ha come parametro un puntatore all'evento, da inviare a chi si è registrato.



Questa libreria di gestione degli eventi è presente in java sotto il nome di `java.awt.event.*`; a questo punto possiamo dare una migliore rappresentazione alla nostra GUI:

```
import javax.swing.*;
import java.awt.event.*;

public class Gui1 implements ActionListener {
 private JFrame frame;
 private JButton pulsante;

 public static void main (String [] args) {
 new Gui1().go();
 }
 public void go() {
 frame = new JFrame();
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 pulsante = new JButton("Cliccami");
 pulsante.addActionListener(this);

 frame.getContentPane().add(pulsante);
 frame.setSize(300, 300);
 frame.setVisible(true);
 }
 public void actionPerformed (ActionEvent event) {
 pulsante.setText("Pulsante cliccato!");
 }
}
```

La sorgente, quando viene cliccata, genera l'evento. L'ascoltatore riceve la notifica di quell'evento e come parametro del metodo di notifica c'è un oggetto che descrive l'evento.

### Utilità delle classi interne

Proviamo ad implementare una GUI con due pulsanti e due azioni senza ricorrere all'uso delle classi interne:

```
public class Gui2 implements ActionListener {
 private JFrame frame;
 private JButton pulsantel, pulsante2;

 // qui il solito codice come prima:
 // - generazione dell'oggetto frame e dei due oggetti pulsanti;
 // - registrazione di this con i due oggetti pulsanti;

 // ora devo descrivere le due diverse azioni:
 public void actionPerformed (ActionEvent event) {
 pulsantel.setText("P1: cliccato!");
 }
 public void actionPerformed (ActionEvent event) {
 pulsante2.setText("P2: cliccato!");
 }
}
```

Questo è però un overloading illegale di metodo poiché non sappiamo distinguere quali dei due metodi è eseguito e quindi quale bottone è stato premuto. A questo punto, potrei definire un unico metodo che chieda all'evento ricevuto come parametro informazioni sulla natura della sorgente e che si comporti in modo diverso in base alla risposta, ma ciò significherebbe costruire codice non ben separato: una modifica del funzionamento della sorgente comporterebbe la modifica di ogni ascoltatore.

Creazione di due classi *ActionListener* separate:

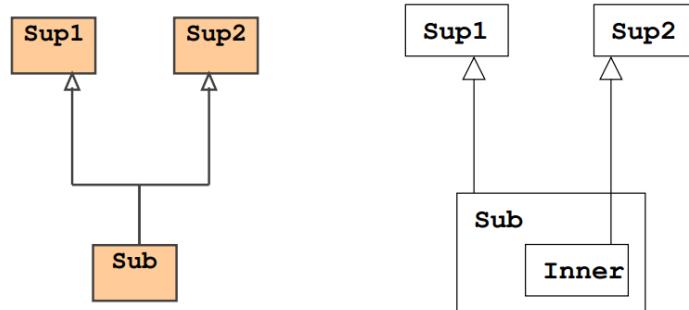
```
public class Gui2 {
 private JFrame frame;
 private JButton pulsante1, pulsante2;
 void go() {
 // codice per istanziare i due oggetti ascoltatori
 // e registrarli con i due oggetti pulsanti
 }
}

class Ascoltatore1 implements ActionListener {
 public void actionPerformed (ActionEvent event) {
 pulsante1.setText("P1 : cliccato !");
 }
}

class Ascoltatore2 implements ActionListener {
 public void actionPerformed (ActionEvent event) {
 pulsante2.setText("P2 : cliccato !");
 }
}
```

Meglio di prima, ma pulsante1 e pulsante2 non sono accessibili all'esterno di Gui2. Per risolvere questo problema avrei diverse soluzioni a disposizione: rompere l'incapsulazione e rendere pubblici i riferimenti ai pulsanti nella classe Gui2, oppure rendere più complicate le classi degli ascoltatori, aggiungendo un attributo che possa riferire al pulsante ed un costruttore che consenta alla classe Gui2 di inizializzare quell'attributo con il valore del proprio attributo (privato) pulsante1 o pulsante2, oppure risolvere tutto elegantemente usando le classi interne.

Le classi interne dette classi **inner** (chiamate anche classi annidate) sono state aggiunte con la JDK 1.1 e danno al programma maggiore chiarezza e lo rendono più conciso. Offrono, inoltre, una soluzione elegante anche al problema realizzativo dell'eredità multipla:



## 52. Classe membro di altra classe

### Introduzione

Le classi interne sono, fondamentalmente, come le altre classi, ma sono dichiarate all'interno di altre classi. Possono essere poste in qualunque blocco, incluso i blocchi dei metodi (con ovvie restrizioni in più). Classi definite all'interno di metodi differiscono leggermente dalle altre e saranno trattate meglio in seguito. Per ora, con il nome di "classe membro", intenderemo una classe che non è definita in un metodo, ma semplicemente in un'altra classe (come gli altri membri: attributi e metodi). La complessità dell'argomento è relativa agli ambiti di validità e di accesso, in particolare all'accesso a variabili nell'ambito includente.

## Sintassi base

```

public class Esterno {
 private int x;
 public class Interno {
 private int y;
 public void metodoInterno() {
 System.out.println ("y = " + y);
 }
 }
 public void metodoEsterno () {
 System.out.println("x = " + x);
 }
 // altri metodi ...
}

```

allo stesso modo con cui una classe appartiene al proprio pacchetto.

È illegale che una classe ed un pacchetto abbiano lo stesso nome, perciò non c'è ambiguità di interpretazione. Attenzione: la rappresentazione puntata vale solo all'interno del sorgente Java; essa non riflette il nome del file delle singole classi. Sul disco la classe interna si chiama *Esterno\$Interno*.

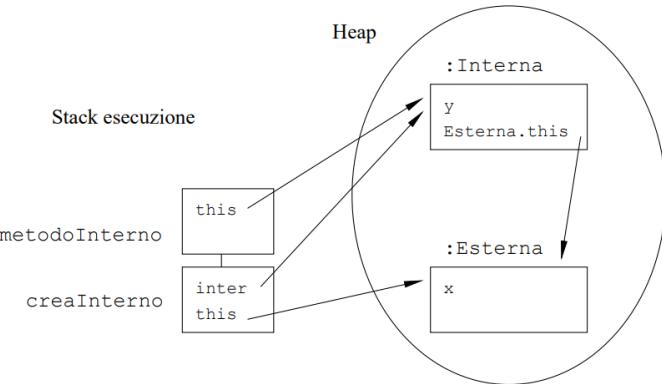
## Costruzione

```

public class Esterna {
 private int x;

 public class Interna {
 private int y;
 public void metodoInterno() {
 System.out.println("x = " + x);
 // attributo esterno accessibile anche con Esterna.this.x
 System.out.println("y = " + y);
 // attributo interno
 }
 }
 public void metodoEsterno () {
 System.out.println("x = " + x);
 }
 public void creaInterno() {
 Interna inter = new Interna();
 inter.metodoInterno();
 }
 // altri metodi ...
}

```



Quando si deve costruire una istanza di una classe Interna, in genere, dove esistere già una istanza della classe Esterna che agisca da contesto. L'oggetto interno può accedere a tutte le componenti dell'oggetto esterno, indipendentemente dalla loro visibilità, attraverso un riferimento implicito. L'oggetto esterno, avendo costruito quello interno, ne possiede un riferimento; l'oggetto interno, però, può accedere all'oggetto esterno attraverso il riferimento implicito (tale riferimento può essere anche esplicitato; nel nostro esempio si chiama *Esterna.this*): oggetti *interno* ed *esterno* si appartengono. A destra dell'immagine un esempio grafico di come si comportano all'interno della memoria.

Nel caso in cui non esista l'oggetto esterno, per esempio perché la JVM sta eseguendo un metodo statico, allora bisogna costruirne uno “al volo”:

Il nome della classe includente diventa parte del nome della classe inclusa, in questo caso si può accedere ad *Interno* anche esternamente dalla classe (essendo pubblica) e i nomi completi delle due classi sono *Esterno* ed *Esterno.Interno*, come se fosse un qualsiasi attributo statico.

Questo formato è reminiscente del modo in cui una classe è nominata all'interno di un pacchetto. Infatti, una classe interna appartiene alla propria classe includente

```

public static void main (String args []) {
 Esterna.Interna i = new Esterna().new Interna();
 i.metodoInterno();
}

```

Costruisce l'oggetto della classe *Interna* e gli fa eseguire la new *Esterna*, apparentemente manteniamo un puntatore *i* solo ad *Interna* ma attraverso this possiamo raggiungere anche l'oggetto della classe esterna. Oppure si può scrivere in forma meno compatta:

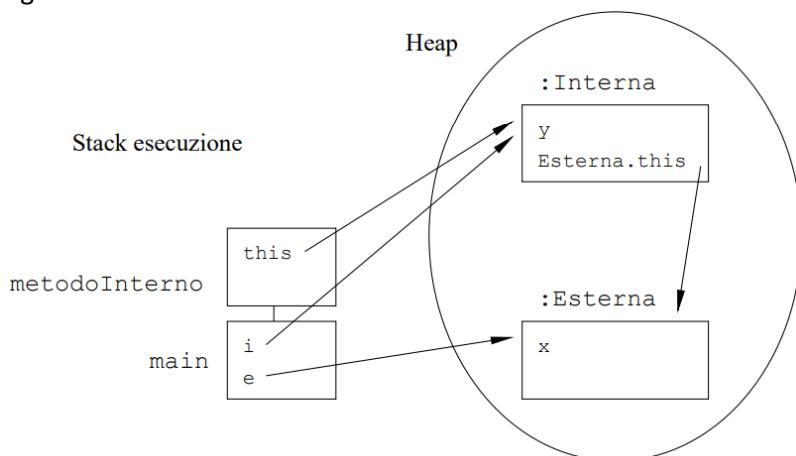
```

public static void main (String args []) {
 Esterna e = new Esterna();
 Esterna.Interna i = e.new Interna();
 i.metodoInterno();
}

```

In entrambi i casi new è usato come se fosse un metodo della classe *Esterna*, poiché, a prescindere dal metodo utilizzato prima di poter creare un oggetto interno bisogna creare l'oggetto *Esterna*.

Esempio grafico:

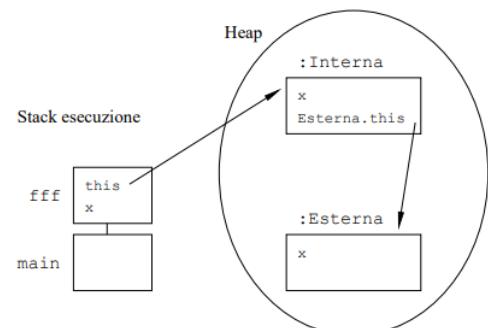


## Esempi

```

public class Esterna {
 private int x;
 public class Interna {
 private int x;
 public void fff (int x) {
 /* parametro locale: */ x++;
 /* attributo interno: */ this.x++;
 /* attributo esterno: */ Esterna.this.x++;
 }
 }
}

```



Applichiamo ora questa semantica all'esempio di GUI a due pulsanti:

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Gui2 {
 private JFrame frame;
 private JButton pulsantel, pulsante2;

 public static void main (String[] args) {
 new Gui2().go();
 }
}

```

```

 public void go() {
 frame = new JFrame();
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 pulsante1 = new JButton("Pulsante 1");
 pulsante1.addActionListener(new Ascoltatore1());

 pulsante2 = new JButton("Pulsante 2");
 pulsante2.addActionListener(new Ascoltatore2());

 frame.getContentPane().add(BorderLayout.WEST, pulsante1);
 frame.getContentPane().add(BorderLayout.EAST, pulsante2);
 frame.setSize(300, 300);
 frame.setVisible(true);
 }

 class Ascoltatore1 implements ActionListener {
 public void actionPerformed(ActionEvent event) {
 pulsante1.setText("P1: cliccato!");
 }
 }
 class Ascoltatore2 implements ActionListener {
 public void actionPerformed(ActionEvent event) {
 pulsante2.setText("P2: cliccato!");
 }
 }
}

```

Ora quando uno dei due pulsanti viene premuto la sua notifica arriverà ad un ascoltatore diverso e di conseguenza non avremo ambiguità come nel codice descritto nel capitolo precedente.

## Modificatori

Classi membro possono essere optionalmente marcate con qualunque modificatore di accesso (a differenza delle classi top-level, che possono essere optionalmente marcate solo `public`). Il significato è identico a quello degli altri membri della classe (attributi o metodi). Per esempio, una classe membro `private` può essere vista solo all'interno della classe includente: non si può nominarla al di fuori della classe includente. Classi membro `final` non possono essere specializzate (non ho sottoclassi). Classi membro `abstract` non possono essere istanziate, ma posso istanziare solo le sue sottoclassi.

Infine, le classi membro `static` esistono a prescindere dell'oggetto includente (sono tutti statici); non hanno il riferimento implicito all'oggetto includente; sono praticamente delle classi top-level, con uno schema di nomi modificato e con un accesso privilegiato per la classe esterna. Sono istanziate in questo modo:

```

public class Esterna {
 public static class Interna{}
 public static void main (String [] args) {
 Interna i = new Esterna.Interna();
 }
}

```

Vogliamo esplicitare che esiste una importante restrizione ai modificatori attribuibili ai membri di una classe interna (attenzione, non ai membri di una classe top-level):

- Poiché una istanza di una classe interna non-`static` può solo esistere insieme ad una istanza della classe includente, cioè, in altri termini, poiché una classe interna non-`static` esiste solo per generare istanze e non per fornire servizi, allora i membri di una classe interna non-static non possono essere marcati static.

- Ciò implica, per esempio, che:
  - possono esistere classi static di livello di annidamento diverso dal primo solo se sono annidate in altre classi static;
  - è possibile che esistano interfacce annidate, ma in questo caso esse hanno l'implicito (o esplicito) modificatore static e seguono la stessa regola precedente.

## 53. Classe definita all'interno di un metodo

### Introduzione

Se creo una classe dentro un metodo l'ambiente non locale non comprende solo gli attributi esterni ma anche quelli interni al metodo. Delle classi definite in un metodo ha le seguenti caratteristiche:

- 1) Come tutte le altre variabili del metodo, sono locali in quel metodo e non possono essere marcate con nessun modificatore tranne che con abstract o final (non entrambi, perché uno è in conflitto con l'altro).
- 2) Esse sono private nel metodo e dunque non sono accessibili (come tutte le altre variabili locali) in alcun modo all'esterno del metodo.
- 3) Hanno accesso limitato alle altre variabili locali del metodo.

### Accesso a variabili locali

La regola di accesso è semplice: *ogni variabile locale (anche un parametro) del metodo includente non può essere utilizzata dalla classe interna, a meno che quella variabile non sia marcata final.*

In realtà un oggetto (sullo heap) non dovrebbe accedere affatto alle variabili dei metodi (di stack), poiché gli oggetti sullo heap possono sopravvivere ai record sullo stack. Appena il metodo finisce le variabili cessano di esistere, quindi se mi rimanesse un oggetto sullo heap esso continuerebbe ad esistere anche dopo che il suo ambiente non locale è distrutto dunque porterà ad errori.

Il trucco utilizzato dalla JVM per permettere l'accesso limitatamente alle variabili final è il seguente: La variabile final è una costante nel metodo. L'oggetto che deve accedervi ne possiede semplicemente una copia (per esempio generata durante la costruzione dell'oggetto). In tal modo, alla terminazione del metodo, la costante sarà ancora utilizzabile, anche se non esiste più il record di attivazione che la contiene.

### Esempio

```

public class Esterna {
 private int m = (int) (Math.random() * 100);
 public static void main (String [] args) {
 Esterna oggetto = new Esterna();
 oggetto.fai(m, 2 * m);
 }

 public void fai (int x, final int y) {
 int a = x + y;
 final int b = x - y;
 class Interna {
 public void metodo() {
 System.out.println("m = " + m);
 // System.out.println("x = " + x); // Illegale
 System.out.println("y = " + y);
 // System.out.println("a = " + a); // Illegale
 System.out.println("b = " + b);
 }
 }
 Interna oggetto = new Interna();
 oggetto.metodo();
 }
}

```

## 54. Classi anonime

### Introduzione

Sostanzialmente servono per estendere un'altra classe (fare overriding di qualche metodo in una classe già esistente) oppure, in alternativa, per implementare una singola interfaccia. La sintassi, però, non concede di fare entrambe le cose, né di implementare più di una interfaccia: se si dichiara una classe che implementa una singola interfaccia, allora la classe è sottoclasse diretta di `java.lang.Object`.

Poiché non si conosce il nome di una tale classe, non si può usare `new` nel modo solito per crearne una istanza. Infatti, la definizione, la costruzione e il primo uso (spesso in una assegnazione) avvengono nello stesso enunciato. L'utilità è nella possibilità di sovrapporre qualche metodo della superclasse (o di implementare metodi di una interfaccia) senza la necessità di scrivere una vera classe che lo faccia.

### Esempio

```
1 class A {
2 public void f () {
3 System.out.println("A");
4 }
5 }
6 class B {
7 A a = new A() {
8 public void f() {
9 System.out.println("B");
10 }
11 }; // notare il ';'
12 }
```

La variabile `a` si riferisce non ad una istanza di `A`, ma ad una istanza di una sottoclasse anonima di `A`.

La linea 7 comincia con una dichiarazione di variabile-riferimento ad `A`. Ma, invece di essere: `A a = new A();` (con `'.'`) c'è una parentesi graffa aperta alla fine, che si chiude alla linea 11. Qui c'è il punto e virgola che manca.

Quello che c'è tra la linea 7 e la 11 si può parafrasare nel seguente modo: "Dichiara una variabile `a` di tipo `A`; poi dichiara una nuova classe, senza nome, sottoclasse di `A`, che contenga solo una sovrapposizione del metodo `f()`; infine, costruisci una istanza di tale sottoclasse ed assegna il suo riferimento ad `a`".

Il polimorfismo è in azione: noi usiamo un riferimento ad una superclasse per riferirci ad una istanza di una sottoclasse. Ciò implica:

- 1) non avremo mai la possibilità di accedere a ciò che si aggiunge nella definizione della sottoclasse anonima (con quale nome faremmo un cast?);
- 2) l'uso di una classe interna anonima è possibile solo per quanto riguarda i metodi che essa sovrappone, e che sono citabili solo perché presenti nella superclasse;
- 3) essa non può avere un costruttore esplicito (che nome avrebbe?).

```
class A {
 public void f () {...}
}
class B {
 A a = new A() {
 public void g () {...}
 public void f () {...}
 };
 public void usa () {
 a.f(); // OK , f() è anche metodo di A
 a.g(); // Illegale, g() non è un metodo di A
 }
}
```

# Paradigma Funzionale; ML

## 55. Introduzione

### Paradigma funzionale

Il paradigma funzionale è quel paradigma dove i programmi sono funzioni e la valutazione di questi programmi si riduce praticamente al calcolo delle espressioni, come linguaggio di esempio descriveremo ML.

Programmare in stile funzionale puro significa usare solo espressioni e funzioni, eventualmente ricorsive. Non vi sono assegnamenti, non c'è una memoria che cambia, perché gli environment mappano gli identificatori direttamente sul loro valore (immutable) invece di una locazione di memoria (il cui contenuto può cambiare); quindi, senza assegnamenti, non ci possono essere cicli while/for e questo ha le seguenti conseguenze sulla programmazione:

- ricorsione al posto dei cicli
- modifiche all'ambiente anziché agli assegnamenti:
- creazione di nuovi identificatori con lo stesso nome che mascherano la versione precedente (come nello scoping statico)

Il linguaggio ML supporta le seguenti implementazioni:

- interprete interattivo (lo useremo per molti esempi)
- implementazione mista (compilazione su codice intermedio e successiva interpretazione)
- compilazione su codice oggetto *standalone*.

### ML

ML è un linguaggio fortemente e staticamente tipato, quindi il controllo dei tipi avviene interamente a tempo di compilazione, ma non richiede di dichiarare il tipo degli identificatori, infatti, spesso lo capisce da solo (*type inference*). Usa sia *structural equivalence* sia *name equivalence*; permette di definire tipi ricorsivi (liste, alberi, ...) e supporta polimorfismo parametrico (come i template). Ha un garbage collector.

Supporta encapsulation (tipi di dato astratti) ma non è un linguaggio a oggetti, quindi mancano la gerarchia di tipi e, di conseguenza, l'ereditarietà.

ML può essere usato in 2 modi:

- 1) Interagendo con l'interprete
  - inserendo definizioni ed espressioni una per una
  - l'interprete risponde ad ogni passo
  - si possono caricare programmi da file digitando nella shell dell'interprete il comando *use "nome del file"*; questo rende utilizzabili le dichiarazioni contenute nel file
- 2) Compilando un programma in codice oggetto direttamente eseguibile
  - ad es. mediante il compilatore mlton per standard ML *mlton "nome del file.sml"* questo comando produce un file eseguibile con lo stesso nome (ma senza l'estensione .sml).

### I tipi primitivi in ML

Detti anche base types:

- **int**: gli interi hanno la solita sintassi eccetto per i negativi che invece del segno – usano la ~: 0, 1, ~1, 2, ~2, ..., 0xff, ~0/x32, ... sono interi. Alcuni operatori: +, -, \*, /, div, mod, =, <, ...
- **word**: le word iniziano con 0w invece di 0x e rappresentano gli interi senza segno (unsigned integers).
- **real**: Solita rappresentazione anche per i real, alcuni suoi operatori sono: +, -, \*, /, =, <, ...
- **string**: rappresentate dentro doppi apici: "stringa", alcuni operatori: ^, size, =, <, ...

- **char**: Singolo carattere rappresentato tra apici precedute con #: # " a ", # "\ n ", # "\ 163", ...
- **bool**: Possono valere o true o false, gli operatori sono *not*, *andalso*, *orelse*, =

Esempi di interazioni con l'interprete ('it' si riferisce all'espressione data; calcola sia valore che tipo):

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- 0w7 mod 0w4 ;
val it = 0wx3 : word

- "Hallo" ^ "world";
val it = "Hallo world" : string

- ord #"a"; ord #"b";
val it = 97 : int
val it = 98 : int

- 3 + 2.2;
Error : operator and operand don't agree [overload conflict]

- real (3) + 2.2;
val it = 5.2 : real
```

Nessuna conversione automatica tra tipi numerici! Usare *real:int->real* e *basis library* (C'è una basis library per ogni tipo primitivo con funzioni per conversioni, parsing, e altre utilità):

```
- val r = 3.0 + 2;
Error : operator and operand don't agree

- val r = 3.0 + real(2);
val r = 5.0 : real

- val i = 1 + 0wl;
Error : operator and operand don't agree

- val i = 1 + Word.toInt(0wl);
val i = 2 : int
```

Real non supporta l'uguaglianza perché lo standard IEEE prevede valori che risultano da operazioni non definite, denominati NaN (not a number), quindi si usa la sintassi *Real.==*:

```
- val x = 1.0; val y = 2.0;
val x = 1.0 : real
val y = 2.0 : real

- x = y;
Error: operator and operand don't agree [equality type required]

- Real .==(x , y);
val it = false : bool
```

Un NaN non è confrontabile con nessun altro numero, nemmeno con sé stesso:

```
- val e = Math.sqrt(~2.0);
val e = nan : real

- Real.==(e, e);
val it = false : bool
```

## 56. Dichiarazione e scoping in ML

### Funzioni

Ci sono diversi modi di definire e chiamare funzioni in ML, i più tradizionali sono: fun è la parola chiave per dichiarare la funzione, segue il nome della funzione e separati da spazi gli argomenti. Di seguito un esempio di una funzione che calcola il fattoriale:

```
- fun fatt x = if x =0 then 1 else x * fatt (x -1);
val fatt = fn : int -> int

- fatt (3);
val it = 6 : int

- fatt 3; (* in questo caso le parentesi sono opzionali *)
val it = 6 : int
```

fun aggiunge all'ambiente l'identificatore fatt e lo associa alla funzione da interi a interi.

Si può vedere cosa è associato a fatt senza chiamare la funzione:

```
- fatt; (* nome della funzione senza argomenti *)
val it = fn : int -> int
```

Mostra solo il tipo (il valore è stato trasformato in bytecode).

### Altre dichiarazioni di identificatori

Con val si aggiunge un nuovo identificatore all'ambiente e gli si associa un valore (in seguito vedremo che val è più generale di fun):

```
- val x = 2+2;
 val x = 4 : int - x+2;
 val it = 6 : int
```

Di seguito la grammatica delle dichiarazioni viste sinora:

```
< declaration > ::=
 val < id name > = < expression > |
 fun < func name > < argument >* = < expression >
```

### Scoping

ML ha i blocchi, la sua struttura è la seguente:

```
let
 < dichiarazioni >
in
 < espressione > (* le dichiarazioni valgono solo qui *)
end
```

Lo scoping è **statico**. Esempi:

```
- let val x=2 in 3*x end ;
val it = 6 : int

- x;
Error: unbound variable or constructor: x

- let val x=2 in
 let val x =3 in (* questa def. maschera la precedente *)
 | 3*x
 end
 end;
val it = 9 : int
```

Esempio di ambiente non locale delle funzioni:

```
- val x=0;
val x =0 : int

- let val x =1 in
 let fun f (y) = x+y in (* x è non locale *)
 ...
 f(0)
end
end;
val it = 1 : int
```

Poiché dopo let possiamo mettere quante dichiarazioni vogliamo il precedente codice può essere rappresentato in una forma equivalente ma più concisa:

```
let
 val x = 1
 fun f(y) = x + y
in
 f(0)
end;
```

Definizioni ausiliarie (locali ad altre definizioni):

```
local
 < dichiarazioni >
in
 <dichiarazione> (* le dichiarazioni sopra valgono solo qui *)
end
```

Simile a let ma dopo in c'è una dichiarazione invece di una espressione da valutare.

## 57. Tipi strutturati in ML

### Prodotti cartesiani

ML permette di definire n-uple semplicemente mettendo i valori tra parentesi, dal punto di vista del tipo il prodotto cartesiano viene indicato con \* mentre con #i si estraе l'i-esimo elemento da una n-upla. Esempi:

```
- (1+1 , "A");
val it = (2, "A") : int * string

- val x = (1, "A" , 3.5);
val x = (1, "A" , 3.5) : int * string * real

- #1(x);
val it = 1 : int

- #2(x);
val it = "A" : string

- #3(x);
val it = 3.5 : real
```

### Record

I record sono insiemi di espressioni  $\langle nome \rangle = \langle valore \rangle$ , sono come le n-pule ma in essi l'ordine non è influente. Si può notare come viene rappresentato il tipo nel seguente esempio:

```
- val r = { nome = "Mario", nato = 1998};
val r = { nato = 1998, nome = "Mario" } : { nato : int, nome : string }
```

Il valore associato al nome  $N$  si estraе con  $\#N$ :

```
- #nome(r); - #nato(r);
val it = "Mario" : string val it = 1998 : int
```

Come già detto in precedenza l'ordine delle coppie non conta, infatti:

```
- { nome = "Mario", nato = 1998 } = { nato = 1998, nome = "Mario" };
val it = true : bool
```

## Dichiarazioni di tipo

ML permette di definire nuovi tipi similmente ai `typedef` del C:

```
- type coord = real * real;
type coord = real * real
```

Il compilatore va però aiutato a stabilire il tipo (si forza ad assegnare la variabile al tipo definito):

```
- val x = (3.0, 4.0); (* senza aiutino *)
val x = (3.0, 4.0) : real * real

- val x : coord = (3.0, 4.0); (* con aiutino *)
val x = (3.0 ,4.0) : coord
```

Non c'è comunque molta differenza poiché i tipi `coord` e `(real * real)` sono compatibili tra loro (*structural equivalence*), infatti posso passare una espressione di tipo `coord` a un parametro di tipo `(real * real)` e viceversa; similmente `coord` è compatibile con ogni altro tipo definito come `(real * real)`, come ad esempio `type coppia = real * real;`.

## Datatypes e costruttori

Con i datatypes si può fare più che dare un nome a un tipo ML; si possono definire **costruttori** per creare data objects, così da creare un tipo incompatibile con tutti gli altri ( | = oppure):

```
- datatype color = red | green | blue;
datatype color = blue | green | red

- val c = red ;
val c = red : color
```

red, green, blue sono costruttori. Definiscono i possibili valori del tipo color.

Somiglia molto alle enum del C, ma è una somiglianza solo apparente; infatti le enumerazioni in C non sono altro che int camuffati:

```
enum color { red , green , blue };
printf("%d%d%d", red, green, blue); // stampa 012
enum color c = red;
if (c == 0) /* then */ {...} else {...}; // esegue il then
c = 10; // nessun errore!!!
```

Invece i datatypes di ML definiscono tipi genuinamente nuovi (nessuna corrispondenza con gli int):

```
- val c : color = 10;
Error : pattern and expression in val dec don't agree

- c = 0; (* c è uguale a 0? *)
Error : operator and operand don't agree
```

red, green, blue sono oggetti completamente nuovi.

Ogni tipo definito con datatype è incompatibile con tutti gli altri tipi (*name equivalence*)

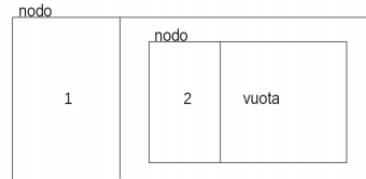
## Costruttori con argomenti

Per dare un esempio di costruttori con argomenti definiamo in maniera **ricorsiva** una lista concatenata di interi, il caso base è rappresentato dalla lista vuota mentre il caso induttivo è un nodo che contiene un intero e una lista di interi.

Quindi servono 2 costruttori, uno per la lista vuota ed uno per i nodi:

```
- datatype listaInt = vuota | nodo of int * listaInt;
datatype listaInt = nodo of int * listaInt | vuota

- val L = nodo (1, nodo(2, vuota));
val L = nodo (1, nodo(2, vuota)) : listaInt
```



Un altro esempio è un albero binario con nodi etichettati da interi. La definizione ricorsiva è un albero vuoto, oppure un nodo che contiene un intero e due alberi dello stesso tipo:

```
datatype albero = vuoto | nodoAlb of int * albero * albero
```

In questo esempio costruiamo un albero con radice 1, figlio sinistro 2 (che è una foglia), mentre il figlio destro manca. `nodoAlb(1, nodoAlb(2, vuoto, vuoto), vuoto)`

## 58. Patterns e Matching

### Utilizzo dei costruttori con argomenti

Per scandire una lista abbiamo innanzitutto bisogno di controllare se è vuota:

```
- val L = nodo (1, nodo(2, vuota));
val L = nodo (1, nodo(2, vuota)) : listaInt

- L = vuota;
val it = false : bool
```

Se non fosse vuota potrebbe servirci il primo elemento, si estrai con *pattern matching*:

```
- val nodo(p,_) = L ; (* assegna a p il 1° elemento di L *)
val p = 1 : int (* "_" è una wildcard *)
```

`nodo(p, _)` è chiamato **pattern**. Per ottenere il resto della lista:

```
- val nodo(,r) = L; (* assegna a r il resto *)
val r = nodo(2, vuota) : listaInt
```

### Funzione che conta gli elementi della lista

La funzione che conta gli elementi della lista deve ovviamente esser ricorsiva (niente cicli):

```
- fun conta x = if x=vuota then 0
 else let val nodo(_,r) = x
 in conta(r) + 1 end;
val conta = fn : listaInt -> int

- conta L ;
val it = 2 : int
```

- Notare come il compilatore ha inferito (derivato) il tipo della funzione
  - 1)  $x$  viene confrontato con "vuota", che è di tipo `listaInt`, quindi anche  $x$  è di tipo `listaInt`, ne consegue che l'input di "conta" è un `listaInt`
  - 2) il "then" restituisce 0, che è un intero; quindi l'output di "conta" è un intero
- Inoltre, il compilatore controlla che anche il resto della funzione sia compatibile con questi tipi
  - 1)  $r$  corrisponde al 2° argomento del nodo, che è di tipo `listaInt` quindi è corretto passarlo a conta che restituisce un intero, di conseguenza anche l'else restituisce un intero

## Definizione per casi

Si possono estrarre tutti gli elementi di un costruttore in un colpo solo dichiarando due identificatori (p e r):

```
- val nodo (p, r) = L;
val p = 1 : int
val r = nodo(2, vuota) : listaInt
```

Si può definire una funzione per casi mettendo direttamente i pattern al posto dei parametri formali:

```
- fun conta(vuota) = 0
| conta(nodo(_, r)) = conta(r) + 1;
```

Un modo analogo per i pattern dello switch/case del C (l'idea è la stessa della definizione per casi delle funzioni):

```
- case L of vuota => true
| nodo (_, _) => false;
val it = false : bool
```

## 59. Liste e Currying

### Le liste in ML

Le liste sono tra le strutture dati più usate in programmazione funzionale, in qualche misura sostituiscono i vettori che sono un concetto essenzialmente imperativo (una sequenza di locazioni di memoria di cui uno può cambiarne il contenuto), invece nei linguaggi funzionali dove la memoria non c'è i vettori sono come delle costanti. Perciò ML fornisce le lista built-in con i costruttori *nil* e *::* (utile per il patching matching)

```
nil (* lista vuota *)
1 :: 2 :: 3 :: nil (* lista che contiene 1, 2, 3 *)

(* formato equivalente basato su parentesi quadre *)
[] (* lista vuota *)
[1 ,2 ,3] * lista che contiene 1, 2, 3 *

(* sono veramente equivalenti *)
- [] = nil ;
val it = true : bool

- 1::2::3:: nil = [1 ,2 ,3];
val it = true : bool
```

### Principali operatori sulle liste in ML

Le liste hanno un certo numero di operatori predefiniti, di seguito si presentano le funzioni più importanti:

- *length* restituisce la lunghezza di una stringa
- *null* restituisce true se la stringa è vuota
- *hd* e *tl* restituiscono rispettivamente il primo elemento della lista e il resto della lista:

```
- val L = [1 ,2 ,3];
val L = [1,2,3] : int list

- hd L;
val it = 1 : int - tl L;
 val it = [2,3] : int list
```

Altre funzioni si trovano nella struttura List (praticamente un package), ad esempio:

- *List.nth(L, i)* restituisce l' *i*-esimo elemento di *L* (partendo da 0)
- *List.last(L)* restituisce l'ultimo elemento di *L*

## Currying

Nel linguaggio ML ogni funzione ha un solo argomento, infatti scrivendo `fun f(x,y) = ...` l'argomento è una (singola) coppia  $x,y$ ; allo stesso modo scrivendo `fun f x y = ...` l'argomento è solo  $x$ , praticamente  $f$  è una funzione che *restituisce una funzione* che prende  $y$  come parametro e calcola l'espressione dopo “=”, di seguito un semplice esempio:

```
- fun f (x, y) = x + y ;
val f = fn : int * int -> int

- fun f' x y = x + y ;
val f' = fn : int -> int -> int
```

Quando ci sono più frecce la parentesizzazione va a destra, quindi il tipo  $\text{int} \rightarrow \text{int} \rightarrow \text{int}$  va inteso come  $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$ .

la trasformazione da n-uple (come  $f(x,y)$ ) a funzioni che restituiscono funzioni (come  $f' x y$ ) si chiama **currying**, ma ovviamente  $f$  ed  $f'$  hanno un utilizzo diverso:

```
- fun f (x, y) = x + y;
val f = fn : int * int -> int - fun f' x y = x + y;
 val f' = fn : int -> int -> int

- f (3 ,2);
val it = 5 : int - f' 3 2; (* viene interpretato come (f'3)(2) *)
 val it = 5 : int

- f 3 2;
Error : operator and operand don't agree - f' (3 ,2);
 Error : operator and operand don't agree

- val g = f' 3; (* significa ce fisso x = 3 quindi f' = 3 + y *)
val g = fn : int -> int

- g 1; (* associo 1 ad y e quindi g = 3 + 1 *)
val it = 4 : int
```

## 60. Funzioni di ordine superiore

### *filter, map, reduce*

Diciamo che il linguaggio funzionale sostituisce i cicli perché la maggior parte delle funzioni ricorsive che operano su liste, alberi e simili hanno la stessa struttura, cambia solo l'operazione che si applica ai nodi. Di conseguenza basta scrivere una volta per tutte la funzione che scandisce la struttura dati (che fa la funzione del ciclo) e passargli la funzione da applicare ai nodi, ovvero quello che bisogna fare all'interno del ciclo.

In questo modo ci ritroviamo ad avere funzioni che prendono altre funzioni come argomento. Le funzioni che hanno altre funzioni come parametri sono dette di **ordine superiore**.

Le tipologie di funzioni/ciclo più comuni sono:

- **filter**: “filtra” una lista, ovvero prende una funzione booleana  $f$  e una lista  $L$  e va a selezionare gli elementi di  $L$  per cui  $f$  è vera:

```
fun filter f [] = []
 | filter f (x::y) = if f(x) then x :: (filter f y)
 else filter f y

(* esempio: seleziona gli elementi negativi da una lista *)
- let fun neg x = x < 0
 in filter neg [0,~1,3,~2] end;
val it = [~1 ,~2] : int list
```

```
(* esempio: seleziona gli elementi positivi da una lista *)
- let fun pos x = x > 0
 in filter pos [0,~1,3,~2] end;
val it = [3] : int list
```

- Mi basta ridefinire la f invece di passargli tutta la ricorsione vista precedentemente.

- **map**: prende una funzione f e una lista L ed applica f a tutti gli elementi della lista

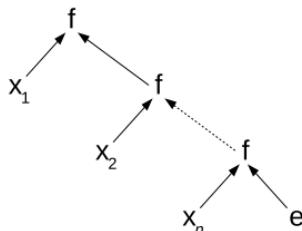
```
fun map f [] = []
| map f(x::y) = f(x)::(map f y)

(* esempio: conversione in lista di reali *)
- map real [1,2,3];
val it = [1.0 ,2.0 ,3.0] : real list

(* esempio: conversione in lista di stringhe *)
- map Int.toString [1,2,3];
val it = ["1","2","3"] : string list
```

- **reduce**: è la più complessa delle tre e serve per calcolare aggregati di una lista (quei valori che dipendono da tutti gli altri di una lista) come ad esempio il minimo, massimo, somma, prodotto, media...
   
Prende in input una funzione  $f$  a 2 argomenti (a differenza della map e la filter che prendono una singola funzione), un valore finale  $e$  (che funge da accumulatore) ed una lista  $L$ . Effettua questo calcolo:

- $\text{reduce } f \ e \ [x_1, x_2, \dots, x_n] = f(x_1, f(x_2, \dots, f(x_n, e) \dots))$



- Ad esempio, se  $f$  è '+' ed  $e = 0$  allora reduce calcola la somma degli elementi della lista.

Definizione di reduce ed esempi:

```
fun reduce f e [] = e
| reduce f e (x::y) = f (x, reduce f e y)

(* esempio sbagliato: somma (+ è infisso) *)
- reduce + 0 [1 ,2 ,3];
Error : ...

(* esempio: somma (corretto) *)
- reduce (op +) 0 [1 ,2 ,3];
val it = 6 : int

(* esempio: media di [x1,...,xn] = x1/n + ... + xn/n *)
- let
 fun f L (elem , accum) = elem / real (length L) + accum
 val lista = [1.0 ,2.0 ,3.0]
 in
 reduce (f lista) 0.0 lista
 end;
val it = 2.0 : real
```

## Funzioni anonime

Spesso quando utilizziamo funzioni di ordine superiori come filter, map e reduce, può far comodo passargli funzioni semplici, specificate lì per lì senza dover dare loro un nome (n'è usare un blocco let), queste funzioni anonime si specificano con la keyword fn, ad esempio:

```
- fn x => x +1; (* x è un parametro non il nome della funzione *)
val it = fn : int -> int

(* per sommare uno a tutti gli elementi di una lista *)
- map (fn x => x +1) [1 ,2 ,3];
val it = [2 ,3 ,4] : int list
```

In Lisp e Scheme l'equivalente di fn è la keyword lambda che storicamente deriva dal lambda calcolo, un modello di calcolo matematico basato su funzioni di ordine superiore a cui tutti i linguaggi funzionali si sono ispirati. Nel lambda calcolo l'operatore  $\lambda$  è l'analogico di fn.

Una funzione anonima non può essere ricorsiva perché non ho un nome di funzione e di conseguenza non posso richiamarla.

Adesso possiamo vedere in che senso val è più generale di fun. Le seguenti definizioni sono equivalenti:

```
- fun f x = x + 1;
val f = fn : int -> int - val f = fn x => x +1;
 val f = fn : int -> int
```

In altre parole, fun è *zucchero sintattico*, cioè una utile abbreviazione per qualcosa che si potrebbe fare in altro modo (con val)

## Ulteriori dettagli su currying

Adesso possiamo mostrare più esplicitamente la natura del currying. Riprendiamo l'esempio  $\text{fun } f' x y = x + y$ ;

In effetti  $f'$  può essere definita equivalentemente come

```
fun f' x = fn y => x+y
val f' = fn : int -> int -> int
```

L'esempio della chiamata di  $f'$  con un solo parametro può essere spiegata come segue:

```
- val g = f' 3; (* come fosse val g = fn y => 3+y *)
val g = fn : int -> int

- g 1;
val it = 4 : int
```

È anche possibile definire funzioni che effettuano il currying e la sua trasformazione inversa per una data funzione del tipo giusto:

```
(* f deve accettare una coppia (x,y) *)
val curry_2args = fn f => fn x => fn y => f (x, y)

(* f deve essere del tipo f x y *)
val uncurry_2args = fn f => fn (x, y) => f x y
```

Esempi di utilizzo:

```
fun f (x, y) = x + y ;
val f' = curry_2args f; (* equivalente a f' x y = x + y *)

(* oppure *)

fun f' x y = x + y ;
val f = uncurry_2args f'; (* equivalente a f (x , y) = x + y *)
```

## 61. Polimorfismo parametrico

### Tipi parametrici

ML supporta l'analogo dei template di C++ e Java, ovvero tipi parametrici ed ML riesce a capire quando un tipo è parametrico, in realtà stiamo usando i tipi parametrici da quando usiamo le liste, ad esempio il costruttore :: può essere applicato a qualunque tipo: `1 :: nil` oppure `"abc" :: nil` ...

```
- length;
val it = fn : 'a list -> int
```

La funzione `length` prende una lista di elementi il cui tipo '`a`' non è specificato, cioè accetta liste con qualsiasi contenuto e, in effetti, non ha bisogno di saperne il tipo: deve solo contare i nodi.

Anche la `map` è parametrica:

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```

Vediamo in seguito come definire tipi parametrici:

```
(* generalizzazione delle nostre liste *)
- datatype 'a lista = vuota | nodo of ('a * 'a lista);
datatype 'a lista = nodo of 'a * 'a lista | vuota

(* alberi binari con etichette parametriche *)
- datatype 'a bt = emptybt | btnode of ('a * 'a bt * 'a bt);
datatype 'a bt = btnode of 'a * 'a bt * 'a bt | emptybt
```

### Type Inference

Per quanto riguarda le funzioni nella maggior parte dei casi ci pensa la type inference a stabilirne il tipo, il compilatore non ha bisogno di alcun aiuto:

```
- fun conta (vuota) = 0
 | conta (nodo (x, l)) = conta (l) + 1;
val conta = fn : 'a lista -> int
```

Come si può vedere dagli esempi precedenti ML usa l'apice prima del nome per indicare che quella è una variabile di tipo, ad esempio '`a` o '`b` o '`c` ...

Quando si vuole che il tipo supporti l'uguaglianza, allora si mette un doppio apice, ad esempio "`a` o "`b` o "`c`..."

Ogni tanto la type inference se ne accorge da sola (ricordarsi che i reali non supportano l'uguaglianza):

```
- fun diag (x , y) = x=y;
val diag = fn : "a * "a -> bool

- diag (1.0 , 1.0);
Error : operator and operand don ' t agree [equality type required]
 operator domain : ''Z * ''Z
 operand : real * real
```

## 62. Encapsulation Interface

### Signatures

Le signature sono il costrutto ML per definire interfacce (nel senso di Java), la nozione di signature definisce tipi e funzioni senza specificarne l'implementazione. Diamo un esempio di implementazione di uno Stack:

```
signature STACK =
sig
 type 'a stack
 val empty: 'a stack
 val push: ('a * 'a stack) -> 'a stack
 val pop: 'a stack -> ('a * 'a stack)
end;
```

il blocco della signature è racchiuso tra le parole chiavi *sig* ed *end*; questo codice dichiara un tipo parametrico *'a stack* senza dire com'è definito; una funzione *empty* per costruire uno stack vuoto, una funzione *push* per inserire un elemento nello stack ed una funzione *pop* per estrarre la testa dallo stack senza, ovviamente, implementarle (per assegnare un tipo a una espressione si usa :).

### Structures

Le structure, come le classi, definiscono i tipi di dato astratti e quindi implementano le signature. Diamo un esempio di implementazione di Stack mediante una lista:

```
structure Stack :> STACK =
struct
 type 'a stack = 'a list;
 val empty = [];
 fun push (x, s) = x::s;
 fun pop (x::s) = (x, s);
end;
```

Con l'espressione *Stack :> STACK* diciamo:

- 1) Stack deve implementare tutti gli identificatori dichiarati in STACK
- 2) I tipi di dato dichiarati in Stack possono essere utilizzati solo con le operazioni dichiarate in STACK
  - ogni altra funzione definita nella structure non è accessibile da fuori
  - così si ottiene *l'encapsulation* e si definiscono tipi di dato *astratti*

### Incapsulamento dell'implementazione dei tipi

Anche se in Stack il tipo stack è implementato con list non si può usare come se fosse di tipo list perché la structure Stack non mette a disposizione alcuna funzione length sul tipo stack e ne nasconde l'implementazione, di conseguenza:

```
type 'a stack = 'a list;

- length [];
val it = 0 : int

- length Stack.empty ;
stdIn:39.1 -39.19 Error : operator and operand don't agree
 operator domain : 'Z list
 operand: 'Y Stack.stack
```

### Functors

È anche possibile dichiarare strutture parametriche, alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri tramite la keyword *functor* (analogo dei template). Di seguito diamo un esempio di definizione di immagini parametriche rispetto alla codifica del colore:

supponiamo di avere due structure RGB e CMYK per gli omonimi modelli di colore e che entrambe implementino la signature COLOR. La struttura parametrica si può dichiarare come segue:

```
functor Image (X : COLOR) =
struct
 (* qui si può usare X come un tipo *)
 (* con le operazioni definite da COLOR *)
end
(* col functor si possono generare diversi tipi di dato *)
structure Image_RGB = Image(RGB);
structure Image_CMYK = Image(CMYK);
```

## 63. Eccezioni e integrazione con type checking

### Eccezioni

Come Java, anche ML ha le sue eccezioni predefinite:  
ma a differenza di Java in ML la gestione delle eccezioni è integrata col type checking:

```
- fun pop(x::s) = (x, s);
Warning: match nonexhaustive
 x::s => ...

- pop [];
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

- 1) la type inference capisce che l'input di pop è una lista
- 2) il datatype lista ha due costruttori: :: e []
- 3) la definizione per casi di pop ha un solo caso per ::
- 4) da cui il warning
- 5) il compilatore inserisce automaticamente una eccezione Match nei casi mancanti

Il programmatore può definire le proprie eccezioni:

```
exception EmptyStack; (* dichiara una nuova eccezione *)

fun pop (x::s) = (x, s)
| pop [] = raise EmptyStack; (* come il throw di Java *)
```

Il risultato in caso di errore è più esplicativo dell'eccezione "automatica" Match

```
- pop [];
uncaught exception EmptyStack
```

### Il costrutto handle

Le eccezioni possono essere catturate e gestite con handle, questo costrutto può essere messo dopo qualunque espressione che può generare una eccezione:

```
pop x
handle EmptyStack =>
 (print "messaggio di errore specializzato";
 raise EmptyStack);

(3 div x) handle ...
```

un singolo handle può gestire diverse eccezioni

```
<expression> handle
 <exception 1> => ...
 | <exception 2> => ...
 | ...
```

A differenza di Java in ML l'ordine in cui si gestiscono le eccezioni è ininfluente poiché non ha oggetti quindi non c'è sovrapposizione (sono tutti oggetti diversi).

In ML funzionale puro sono due i modi di usare il costrutto handle e sono anche le due uniche opzioni che passano il type checking senza errori:

- 1) fare qualcosa come stampare un messaggio e rilanciare l'eccezione
- 2) "aggiustare" l'errore restituendo un valore dello stesso tipo dell'espressione che ha sollevato l'eccezione

Esempio dell'opzione 2: Supponiamo di avere una funzione che restituisce la posizione di  $x$  nella lista, ma se non trova  $x$  solleva un'eccezione (la seguente funzione è a solo scopo didattico, si potrebbe obiettare che sia realizzata male, infatti, manca un caso terminale per  $[]$ ).

```
- fun pos x (y::z) = if (x = y) then 1 else 1 + pos x z;
```

Si può usare handle per modificare  $pos$  per restituire  $-1$  quando non trova  $x$  nella lista:

```
- fun pos2 x y = (pos x y) handle Match = > ~1;
```

### Eccezioni con Parametri

Si possono aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni:

```
(* Esempio di eccezione con parametri: *)

exception SyntaxError of string

(* Questa eccezione può essere lanciata in diversi modi... *)

raise SyntaxError " Identifier expected "
raise SyntaxError " Integer expected "

(* ... e il parametro "letto" col pattern matching *)

... handle SyntaxError x = > ... (* qui si può usare x *)
```

## 64. Esempio: un semplice compilatore

### Elaborazione di simboli in ML

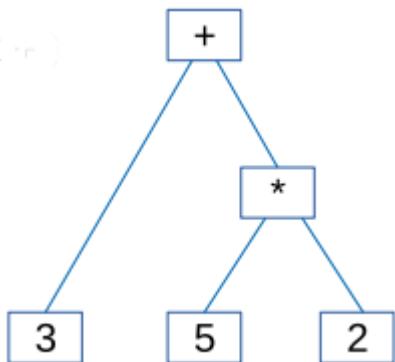
ML, come gli altri linguaggi dichiarativi (non imperativi), è particolarmente adatto alla *manipolazione di simboli*, ed un compilatore fa proprio questo: elabora espressioni e comandi di un linguaggio di programmazione (codice sorgente) e li traduce in un altro linguaggio (codice oggetto o intermedio).

Ne approfittiamo per dare un'idea parziale di alcune strutture dati interne al compilatore e dei procedimenti di generazione e ottimizzazione del codice; l'esempio che segue realizza un compilatore per un linguaggio molto semplificato che supporta solo semplici espressioni su numeri interi. Il codice oggetto deve calcolare l'espressione data.

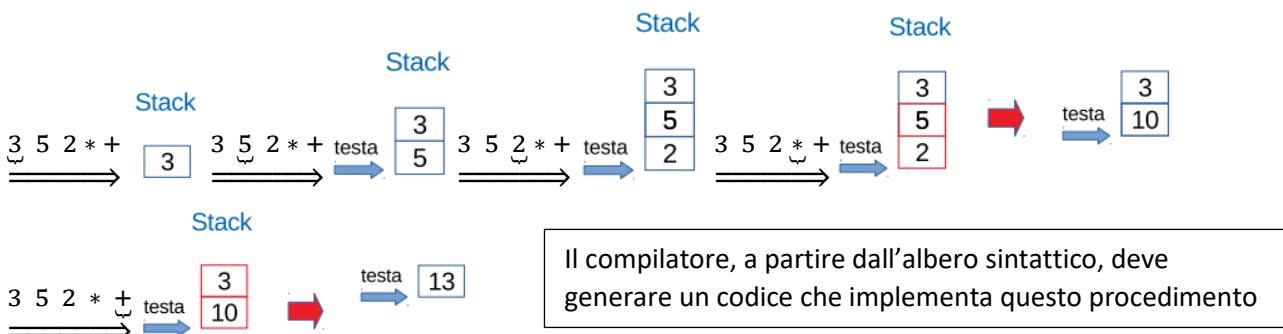
Prima di proseguire bisogna capire il procedimento di valutazione delle espressione (una espressione è nient'altro che una stringa), prendiamo ad esempio l'espressione  $3 + 5 * 2$  che il **parser** trasforma in un albero sintattico (\* ha precedenza su +).

Procedimento di calcolo:

- Salvare i risultati intermedi (come  $5 * 2$ ) su un piccolo stack
- Prima di applicare un operatore bisogna aver calcolato i suoi figli
- Visitando l'albero in ordine posticipato si ottiene l'ordine giusto di valutazione
  - Se sono su una foglia la metto sullo stack
  - Se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi



Esempio di ordine posticipato di visita:  $3 \ 5 \ 2 \ * \ +$  (forma polacca inversa che ha la seguente proprietà: mette gli elementi in ordine da sinistra a destra nello stack, appena arriva ad un operatore prende i primi due argomenti dello stack e li sostituisce con il risultato):



## Definizione del compilatore

Definizione dell'albero sintattico:

Introduciamo un costruttore per ogni operazione supportata dal linguaggio sorgente, ogni costruttore corrisponde a un tipo di nodo dell'albero sintattico

```

datatype syntree = co of int (* le costanti *)
| plus of syntree * syntree
| minus of syntree * syntree
| times of syntree * syntree
| divide of syntree * syntree
| modulus of syntree * syntree

```

Definizione del linguaggio target:

Ovvero le operazioni della macchina astratta che eseguirò il codice oggetto sorgente, qui ci ispiriamo alle istruzioni di hardware classico

```

datatype instruction =
 LOADC of int * int (* LOADC i c => Ri := c *)
| LOADI of int * int (* LOADI i j => Ri := mem (Rj) *)
| STOREI of int * int (* STOREI i j => mem (Rj) := Ri *)
| INCR of int (* INCR i => Ri := Ri + 1 *)
| DECR of int (* DECR i => Ri := Ri - 1 *)
| SUM of int * int (* SUM i j => Ri := Ri + Rj *)
| SUB of int * int (* SUB i j => Ri := Ri - Rj *)
| MUL of int * int (* MUL i j => Ri := Ri * Rj *)
| DIV of int * int (* DIV i j => Ri := Ri / Rj *)
| MOD of int * int (* MOD i j => Ri := Ri mod Rj *)
| HALT

```

## Generazione del codice

Il codice oggetto utilizza due registri:

- R1 come puntatore alla testa dello stack
- R2 funge da accumulatore per operare sul calcolo delle singole operazioni

La traduzione vera e propria è effettuata da una funzione ausiliaria translate che prende in input:

- un albero sintattico tree
- una *continuazione*, ovvero il codice da eseguire dopo avere eseguito le operazioni contenute in tree

Pertanto, la prima chiamata a translate gli passerà l'albero sintattico dell'intera espressione da compilare e come continuazione la lista di istruzione [HALT]

Una buona idea è dare una interfaccia esterna che prende solo un albero sintattico come parametro:

```
fun codegen tree =
let
 (* definizione della funzione translate *)
 (* R1: stack pointer; R2: accumulator *)
 fun translate (co x) cont =
 LOADC (2 , x) ::: INCR (1) ::: STOREI (2 ,1) ::: cont
 | translate (plus(t1,t2)) cont =
 translate t1 (
 translate t2 (
 LOADI(2,1):::DECR(1):::LOADI(3,1):::SUM(2,3):::STOREI(2,1):::
 cont))
 | translate (times(t1,t2)) cont = simile ma con MUL al posto di SUM
 | translate (minus(t1,t2)) cont = simile ma con SUB(3,2) al posto di SUM(2,3)
 | translate (divide(t1,t2)) cont = simile ma con DIV al posto di SUB
 | translate (modulus(t1,t2)) cont = simile ma con MOD al posto di SUB
in
 translate tree [HALT]
end
```

## Ottimizzazione del codice

La generazione meccanica introduce operazioni inutili, infatti il nostro codice generato per  $3 + 5 * 2$  ha delle ridondanze (vedi immagine a destra).

A questo punto aggiungiamo una funzione optimize che elimina le più comuni operazioni ridondanti ed itera una funzione ausiliaria opt1 che esegue un singolo passo di ottimizzazione. Questo può attivare ulteriori semplificazione, infatti, optimize itera opt1 finché il codice non può essere ulteriormente ridotto:

```
local
 (* definizione singolo passo di ottimizzazione *)
 fun opt1 [] = []
 | opt1 (INCR(x):::STOREI(y,_):::DEC(x1):::cont) =
 let val cont' = opt1 cont in
 if x = x1 andalso x <> y then STOREI(y,_):::cont'
 else INCR(x):::STOREI(y,_):::DEC(x1):::cont'
 end
 | opt1 (STOREI(x,y):::LOADI(x1,y1):::cont) =
 let val cont' = opt1 cont in
 if x = x1 andalso y = y1 then STOREI(x,y):::cont'
 else STOREI(x,y):::LOADI(x1,y1):::cont'
 end
end
```

```
LOADC 2 3
INCR 1
STOREI 2 1
—
LOADC 2 5
INCR 1
STOREI 2 1
—
LOADC 2 2
INCR 1
STOREI 2 1
—
LOADI 2 1
DECR 1
LOADI 3 1
MUL 2 3
STOREI 2 1
—
LOADI 2 1
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
—
HALT
```

```

| optl (c::cont) = c::(optl cont) in fun optimize code =
| let val code' = optl code in
| if length(code') = length(code) (* fa 1 passo di ottimizzazione *)
| then code' (* se nessun progresso *)
| else optimize code' (* termina *)
| end
end

```

Il risultato di questa ottimizzazione (per la solita espressione  $3 + 5 * 2$ ) è un guadagno del 30% di istruzioni, quindi il tempo di esecuzione è notevolmente ridotto (a destra il codice ottimizzato).

```

LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
INCR 1
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
MUL 2 3
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT

```

```

LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
LOADI 3 1
MUL 2 3
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT

```

## Combinare le fasi con composizione di funzioni

L'operatore  $\circ$  denota la composizione di funzioni  $(f \circ g)(x) = f(g(x))$ . Con la composizione è facile definire l'intero processo di compilazione assemblando le diverse fasi:

```

- val compile = optimize o codegen o parse;
 val it = fn : string -> instruction list

```

## Conclusioni

La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare. In Java, che pure è un ottimo linguaggio, ogni nodo dell'albero sintattico sarebbe un oggetto e “leggere” la struttura di pezzi di albero non sarebbe immediato.

Inoltre, la type inference ci permette di omettere il tipo degli identificatori, producendo un codice più snello come fosse uno scripting language debolmente tipato, ma senza sacrificare il controllo di tipi forte.

Per queste ragioni linguaggi come ML vengono utilizzati per la prototipizzazione rapida di compilatori e interpreti.

# Paradigma Logico; Prolog

## 65. Introduzione

### Paradigma logico

Un linguaggio logico è un insieme di assiomi, o regole, che definiscono delle relazioni tra oggetti. L'esecuzione di un programma logico non è altro che la derivazione di una conseguenza del programma. Gli assiomi sono espressi nel linguaggio logico tramite strutture dette predicati.

I predicati logici sono un modo semplice ed elegante per esprimere dei ragionamenti in termini logici. È l'utente che stabilisce il significato dei fatti nel mondo reale, ossia che crea l'isomorfismo tra sistema reale e sistema formale.

L'essenza del paradigma logico sono i programmi (insieme di assiomi) e le computazioni, ovvero dimostrazioni costruttive di una formula logica data, detta query, mediante gli assiomi del programma.

### Prolog (PROgramming in LOGic)

L'implementazione in Prolog è analoga a quella di ML, infatti si ha un implementazione mista dove i programmi vengono compilati in un bytecode che viene interpretato da una macchina virtuale chiamata la Warren abstract machine (WAM). Anche l'interazione con Prolog è analoga a quella con ML: si inviano query all'interprete e si ottengono le relative risposte oppure, se il programma è stand alone, si interagisce mediante la sua UI (user interface).

Sistema consigliato per il corso è SWI Prolog (free) che implementa il Prolog standart e supporta sia interpretazione che compilazione stand-alone. Si invoca da command line con il comando *swipl*. Per caricare un proprio programma mioprog.pl si usano i seguenti comandi (? – è il prompt dell'interprete, mentre i commenti sono susseguiti da % oppure racchiusi tra /\* \*/):

```
? - ['mioprog.pl']. % oppure
? - consult('mioprog.pl'). % quando si carica la prima volta;
? - reconsult('mioprog.pl'). % quando si ricarica dopo una correzione
```

I costrutti di base:

- Tre tipi di statement:
  1. fatti (facts)
  2. regole (rules)
  3. queries (detti anche goals)
- Una sola struttura dati:
  1. termini logici (logical terms)

## 66. Costrutti Prolog

### Fatti

I fatti (anche detti assiomi) asseriscono una relazione tra oggetti: *father(abraham, isaac)*. (Abramo è il padre di Isacco); father, oltre che relazione, è chiamato anche predicato e devono iniziare per lettera minuscola, anche gli argomenti abraham e isaac iniziano con lettera minuscola perché sono costanti.

Con i fatti possiamo definire un database (tutto ciò che non viene detto per assiomi il compilatore non lo sa). Diamo un semplice esempio di programma logico creando un database di una famiglia dove ogni predicato corrisponde a una tabella relazionale:

```

father(terach, abraham).
father(terach, nanchor).
father(terach, haran).
father(abraham, isaac).
father(haran, lot).
father(haran, milcah).
father(haran, yiscah).

mother(sarah, isaac).

male(terach).
male(abraham).
male(nanchor).
male(haran).
male(isaac).
male(lot).

female(sarah).
female(milcah).
female(yiscah).

```

## Queries

I programmi logici sono fatti per rispondere a queries. Se il programma caricato è quello descritto precedente allora:

```
? - father(abraham,isaac).
true.
```

```
? - father(isaac,abraham).
false.
```

Le query hanno la stessa forma dei fatti, sono entrambi dei cosiddetti atomi logici che nei programmi sono asserzioni (fatti) mentre nelle query sono domande.

Nel precedente esempio, l'interprete trova il primo atomo nel programma e non trova il secondo e, perciò, da risposte differenti.

Usualmente è utile chiedere cose come: "quali sono i figli di abraham?" e questo si può fare grazie alle variabili logiche che si riconoscono perché iniziano con una lettera maiuscola:

```
?- father(abraham, X). /* esiste X tale che father(abraham,X) ? */
X = isaac.

?- father(terach, X). /* esiste X tale che father(terach,X) ? */
X = abraham;
X = nanchor; % con il punto e virgola chiedo alla macchina
X = haran. % virtuale di darmi un'altra risposta
```

La macchina virtuale cerca i valori che, sostituiti a X, rendono la query uguale a uno dei fatti nel programma.

## Variabili logiche

Le variabili logiche in Prolog differiscono dalle variabili degli altri paradigmi:

- Le variabili logiche rappresentano oggetti qualsiasi (non specificati)
- Le variabili dei linguaggi imperativi sono locazioni di memoria
- Gli identificatori dei linguaggi funzionali denotano valori immutabili

Precedentemente abbiamo mostrato il comportamento delle variabili logiche usate nelle query, ma queste si possono usare anche nei fatti e in questo caso rappresentano un universale. Un credente potrebbe scrivere in un programma: *creato\_da(X,dio)*. e questo fatto sta a significare che ogni cosa è creata da Dio.

Praticamente mentre nelle query le variabili logiche sono esistenzialmente quantificate (esiste un X tale che ...?), nei fatti sono universalmente quantificate (per ogni X vale che ...).

## I termini

I termini sono l'unica struttura dati in Prolog e si costruiscono con costanti, variabili logiche e funtori, quest'ultime si comportano come i costruttori di ML e sono caratterizzati da nome e arietà (il numero di argomenti):

```
< term > ::= < constant > | < variable >
 | < functor name > (<term1>, ..., <termN>)
```

Non ci sono dichiarazioni di tipo. Costanti simboliche e funtori si usano senza dichiararli prima. Esempi:

```
successor(Int) date(25, april, 2020) color(rgb, 0, 0, 1)
```

Gli argomenti di un predicato possono essere termini qualsiasi, quindi la sintassi generale dei fatti è:

```
<fact> ::= <atomic formula>.
<atomic formula> ::= <predicate> (<term1, ..., termN>)
```

Diamo un esempio di query:

```
% fatti
born(john, date(10, october, 2000)).
hasColor(object1, color(rgb, 1, 0, 0)).

?- born(X, date(Y, october, 2000)). ?- hasColor(object1, Y).
X = john, Y = color(rgb, 1, 0, 0).
Y = 10.
```

Diversamente da ML, dove non posso usare la stessa variabile più volte nello stesso pattern perché i termini servono solo ad estrarre informazioni da una struttura, in Prolog la stessa variabile  $X$  può comparire più volte nello stesso fatto o nella stessa query. Significa semplicemente che i termini nelle posizioni dove si trova  $X$  devono essere uguali tra loro.

Un termine che non contiene variabili è detto **ground**, altrimenti è **nonground**. Gli stessi aggettivi e gli stessi criteri si applicano ai fatti, alle query e anche alle regole.

## 67. Matching tra query e fatti (sostituzione e unificazione)

### Sostituzioni e istanze

Una sostituzione è un insieme finito di coppie  $\theta = \{X_1 = t_1, \dots, X_n = t_n\}$  dove:

- Le  $X_i$  sono variabili e i  $t_i$  termini
- Le  $X_i$  sono tutte diverse tra loro
- Nessuna delle  $X_i$  compare dentro i  $t_i$ .

L'applicazione di  $\theta$  ad una espressione  $E$  (che potrebbe essere un termine, un fatto, una query o una regola) si denota con  $E\theta$  e sostituisce le occorrenze delle variabili  $X_i$  in  $E$  con i rispettivi termini  $t_i$ .

Esempio: se  $\theta = \{X = isaac\}$  ed  $E = father(abraham, X)$  allora  $E\theta = father(abraham, isaac)$ .

L'applicazione di una sostituzione  $\theta$  a  $E$  crea un “caso particolare” di  $E$ , dove le variabili di  $E$  (che indicano oggetti non specificati) vengono sostituite con valori specifici (termini ground) o parzialmente specificati (termini non ground).

Esempio:  $\theta = \{X = color(rgb, Y, Y, Y)\}$  e  $E = hasColor(o1, X)$ .

- $E\theta = hasColor(o1, color(rgb, Y, Y, Y))$
- $E$  dice che  $o1$  ha un colore non specificato
- $E\theta$  lo specifica parzialmente: è in formato  $rgb$  e tutti i 3 valori sono uguali

Definizione di istanza:  $E_1$  è un'istanza di  $E_2$  se esiste  $\theta$  tale che  $E_1 = E_2\theta$ , cioè se  $E_1$  è un “caso particolare” di  $E_2$  dove alcune variabili di  $E_2$  sono istanziate, cioè legate a un valore.

### Unificazione

L'algoritmo di unificazione è quello che effettua il matching. Prende due espressioni  $E_1$  ed  $E_2$  e, se possibile, restituisce una sostituzione  $\theta$ , detta **unificatore** (unifier), tale che  $E_1\theta = E_2\theta$ .

L'unificatore costruito dall'algoritmo viene detto *most general unifier* (mgu) perché non sostituisce una variabile con un termine se non è necessario, cioè vincola il meno possibile li risultato  $E_1\theta$ , lasciando, quando

può, le variabili libere. Tecnicamente, ogni altro unificatore  $\theta'$  porta a una istanza (un caso particolare) di  $E_1\theta$ , cioè esiste una sostituzione  $\sigma$  tale che  $(E_1\theta)\sigma = E_1\theta'$ .

Esempi:

- $mgu(sum(A,B,C), sum(X,0,X)) = \{A = X, B = 0, C = X\}$
- $mgu(sum(0,X,0), sum(Y,0,Y)) = \{X = 0, Y = 0\}$
- $mgu(sum(0,X,0), sum(Y,Z,1))$  non esiste a causa del terzo argomento (non si può rendere 0=1)

Nei programmi visti sinora (che consistono di soli fatti) le risposte a una query  $q(t_1, \dots, t_n)$  vengono costruite così (Prolog dice *false* quando nessun fatto unifica con la query):

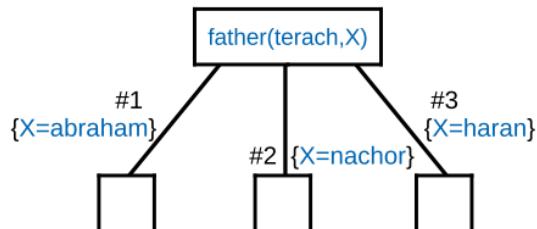
- 1) si cerca nel programma il primo fatto  $p(u_1, \dots, u_m)$  con lo stesso funtore (cioè  $p = q$  e  $m = n$ ). Se se ne trova uno:
- 2) si calcola  $\theta = mgu(q(t_1, \dots, t_n), p(u_1, \dots, u_m))$
- 3) se  $\theta$  esiste, allora:
  - si restituisce  $\theta$  come risposta (se è vuota allora Prolog dice *true*)
  - poi se l'utente non vuole altre soluzioni si termina
- 4) altrimenti si cerca il prossimo fatto con lo stesso funtore. Se esiste si ripete da 2, altrimenti si termina

## Search tree e Conjunctive queries

La rappresentazione grafica del procedimento di unificazione è data dal search tree, diamo un esempio per la query  $father(terach, X)$ :

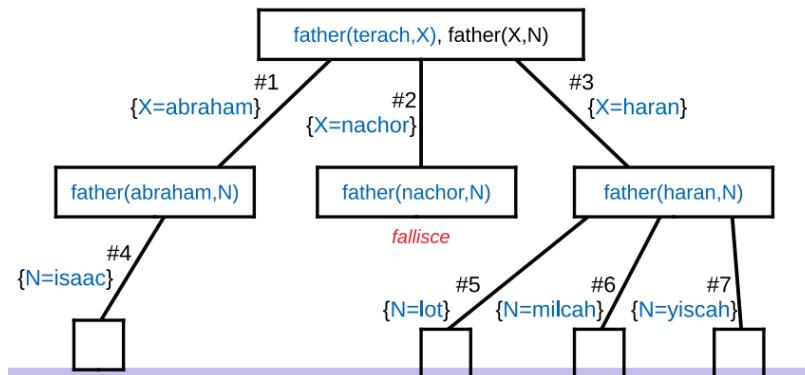
```
/* programma */
/*1*/ father(terach, abraham).
/*2*/ father(terach, nachor).
/*3*/ father(terach, haran).
/*4*/ father(abraham, isaac).
...

```

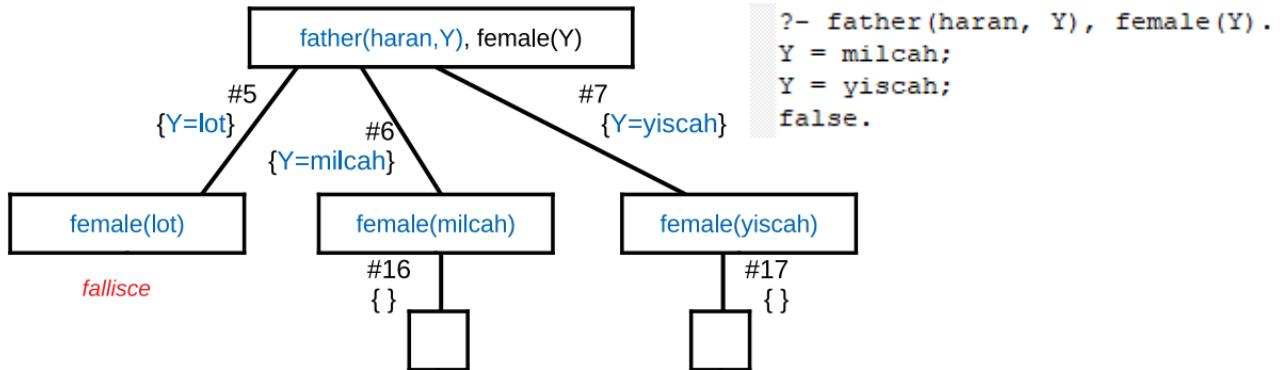


- Gli archi corrispondono ai fatti che unificano con la query
- Sono etichettati con
  - il numero del fatto utilizzato (nell'ordine in cui compare nel programma)
  - il mgu della query e del fatto
- Il search tree di una query (rispetto a un programma) comprende tutte le risposte che Prolog genera se l'utente glielo chiede
- Le computazioni di Prolog corrispondono a una visita depth-first da sinistra a destra (ogni ramo di successo, ovvero un ramo che non fallisce, rappresenta una risposta)

Le query possono contenere più formule atomiche (goals) e in questo caso vengono dette query congiuntive, quest'ultime sono infatti una composizione, complessa o meno, di goal. Ad esempio, possiamo scrivere  $father(terach, X), father(X, Nipote)$  che rappresenta la domanda “Chi sono i nipoti di terach?”, (la virgola è un and) è come un join:



Di conseguenza le figlie di haran le trovo con la query congiuntiva  $\text{father}(\text{haran}, Y), \text{female}(Y)$  dove i valori di  $Y$  mi danno le figlie di haran:



## 68. Ragionamento

### Le regole

Un **programma logico** è un insieme di regole (con e/o senza corpo) per definizione. In generale la sintassi delle regole è la seguente:

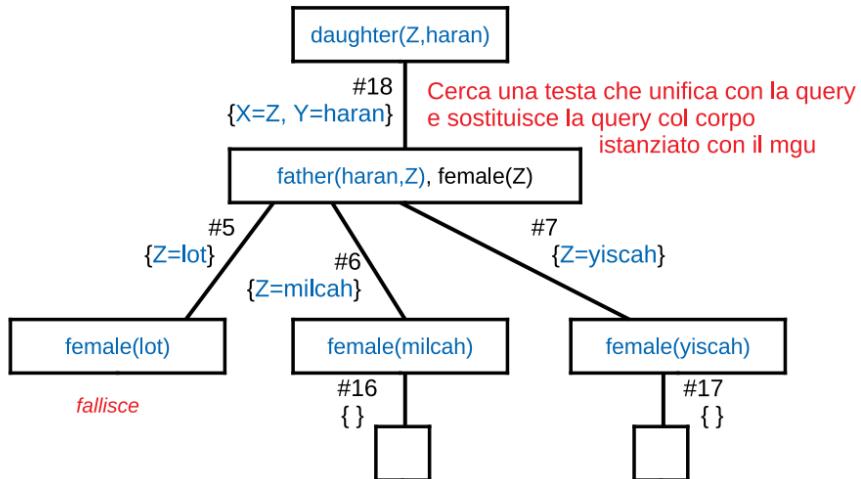
```

<rule> ::= <atomic formula> :- <goal list>.
<goal list> ::= <atomic formula>, ..., <atomic formula N>

```

La formula atomica prima di `:` – è detta **testa** (head), la parte dopo è chiamata **corpo** (body), notare che i fatti non sono altro che regole senza corpo. A questo punto possiamo definire la regola *daughter* poiché sappiamo che  $X$  è figlia di  $Y$  se  $Y$  è padre di  $X$  e  $X$  è femmina:  $\text{daughter}(X, Y) :- \text{father}(Y, X), \text{female}(X)$ .

Aggiungiamo in coda al programma della famiglia biblica (posizione 18) la regola che definisce cos'è una figlia. Di conseguenza le risposte alla query  $\text{daughter}(Z, \text{haran})$  si trovano nel seguente modo:



Quando si ragiona con le regole bisogna fare attenzione con le variabili: quelle della query devono sempre essere diverse da quelle delle regole, se così non fosse si avrebbe il seguente problema:

```

daughter(X, Y) :- father(Y, X), female(X).

?- daughter(Y, haran)
/* le risposte verrebbero perse */

```



Per evitare questo problema, ogni volta che una regola viene usata la WAM ridenomina le sue variabili, ad esempio  $\text{daughter}(_{101}, _{102}) :- \text{father}(_{102}, _{101}), \text{female}(_{101})$ . così da essere sicuro di non avere alcuna variabile in comune con il goal attuale.

Con la definizione attuale, però, ci perdiamo le figlie delle donne. Come soluzione si potrebbe aggiungere un'altra regola in fondo al programma che mi permette di dedurre che  $X$  è figlia di  $Y$  quando  $Y$  è madre di  $X$  e  $X$  è femmina, ovvero:  $daughter(X, Y) :- \neg mother(Y, X), female(X)$ .

Oppure si potrebbe introdurre il concetto di genitore (parent):

```
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

/* può essere riutilizzato per diverse definizioni */

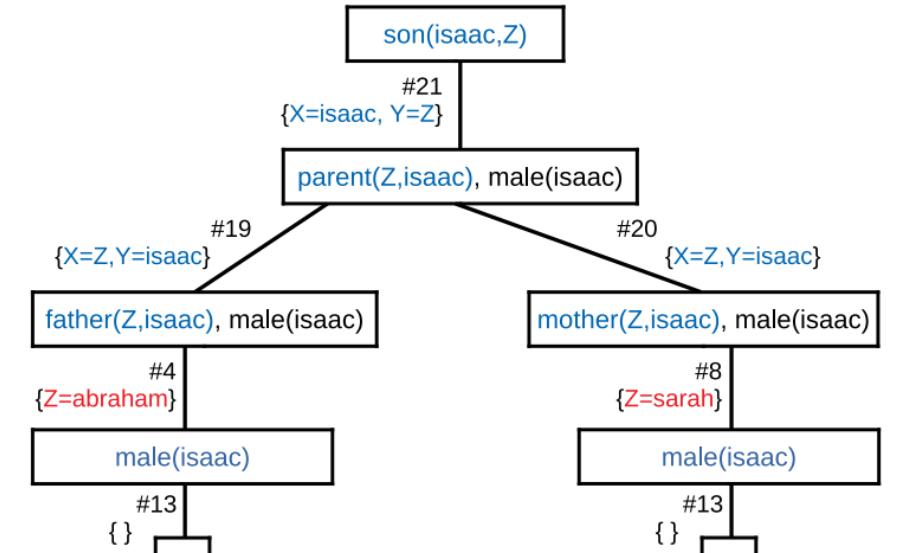
daughter(X, Y) :- parent(Y, X), female(X).
son(X, Y) :- parent(Y, X), male(X).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

## Derivazioni

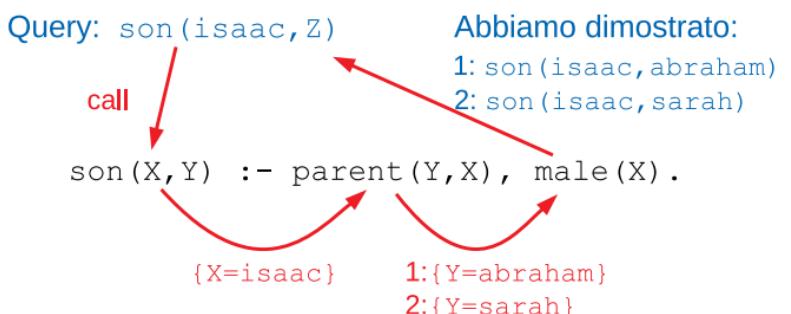
Riprendiamo il programma sulla famiglia biblica:

```
/* 4 */ father(abraham,isaac). ...
/* 8 */ mother(sarah,isaac). ...
/*13*/ male(isaac). ...
/*19*/ parent(X,Y) :- father(X,Y).
/*20*/ parent(X,Y) :- mother(X,Y).
/*21*/ son(X,Y) :- parent(Y,X), male(X).
```

Vediamo che forma prendono gli alberi con queste derivazioni: per la query  $\neg son(isaac, Z)$ :



Ovviamente il primo risultato sarà  $Z = abraham$  poiché non essendoci altro da dimostrare (si raggiunge un goal vuoto) Prolog restituisce la soluzione. Se chiediamo altre soluzioni (con punto e virgola) si farà backtrack da  $male(isaac)$ , non trovando altri match ritorna ancora indietro fino a  $parent(Z, isaac)$  che questa volta ha un altro match:  $Z = sarah$ . Un altro modo di vedere le derivazioni è il seguente:



## Overloading e Wildcards

Si può usare lo stesso nome di predicato con numeri diversi di argomenti e, anche se predici simili, vengono trattati come predici diversi, ad esempio dall'informazione che "X è madre di Y" si può derivare il concetto di essere madre *mother(Mom)*:  $-mother(Mom, X)$ , ovvero *Mom* è una mamma se esiste un *X* tale che *Mom* è madre di *X* (notare che abbiamo un *mother* unario e uno binario). Poiché nel goal *mother(Mom, X)* il valore di *X* non interessa si possono usare wildcards simili a ML: *mother(Mom)*:  $-mother(Mom, \_)$ .

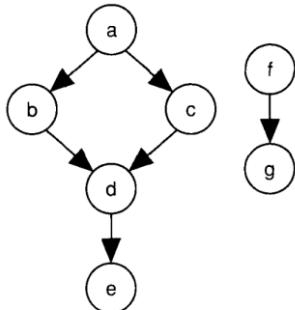
Attenzione: se usiamo due volte una variabile (ad es. *X*) allora in quei punti devo avere lo stesso valore, mentre ogni wildcard può essere associata a un valore diverso:

```
?- mother(X, X).
false /* nessuna è madre di se stessa */

?- mother(_, _).
true /* cerca un fatto mother(X, Y) nel database */
```

## Regole ricorsive

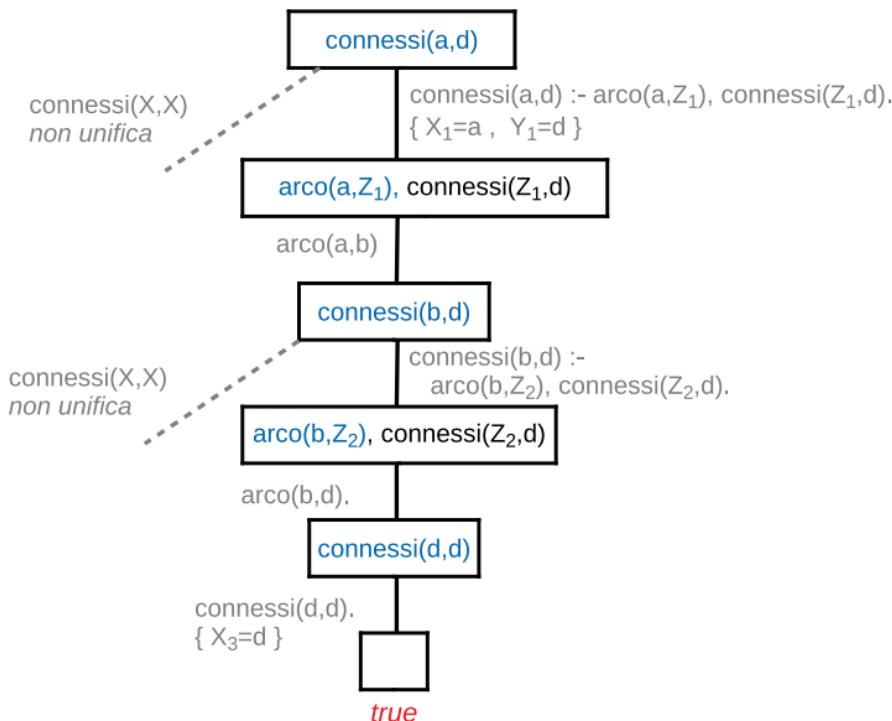
Cerchiamo i cammini in un grafo:



```
/* programma che descrive il grafo */
arco(a, b). arco(a, c).
arco(b, d). arco(c, d).
arco(d, e). arco(f, g).

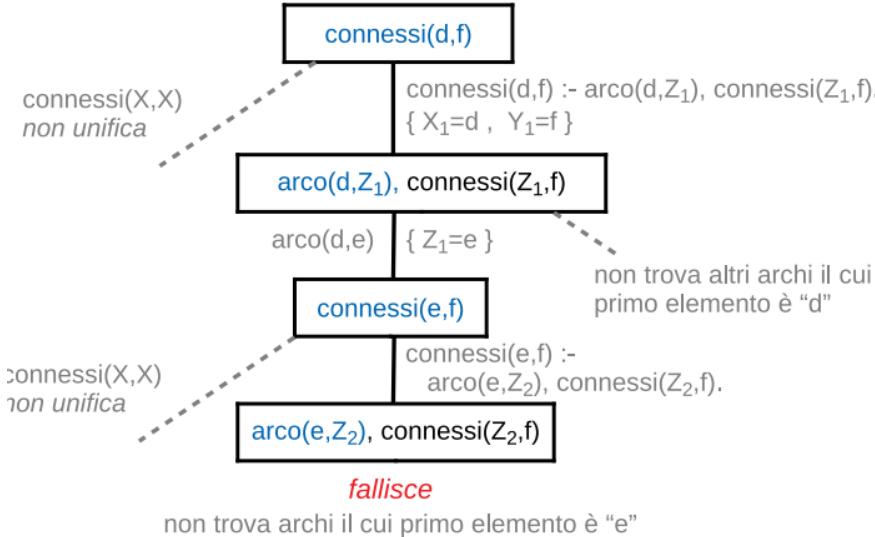
/* verificare se due nodi sono connessi */
connessi(X, X).
connessi(X, Y) :- arco(X, Z), connessi(Z, Y).
```

Search tree parziale per la query *connessi(a, d)*:



Non cerca altre soluzioni perché la query è ground (inutile restituire altri "true").

Mentre se cercassimo la query `connessi(d,f)` la risposta sarà false:



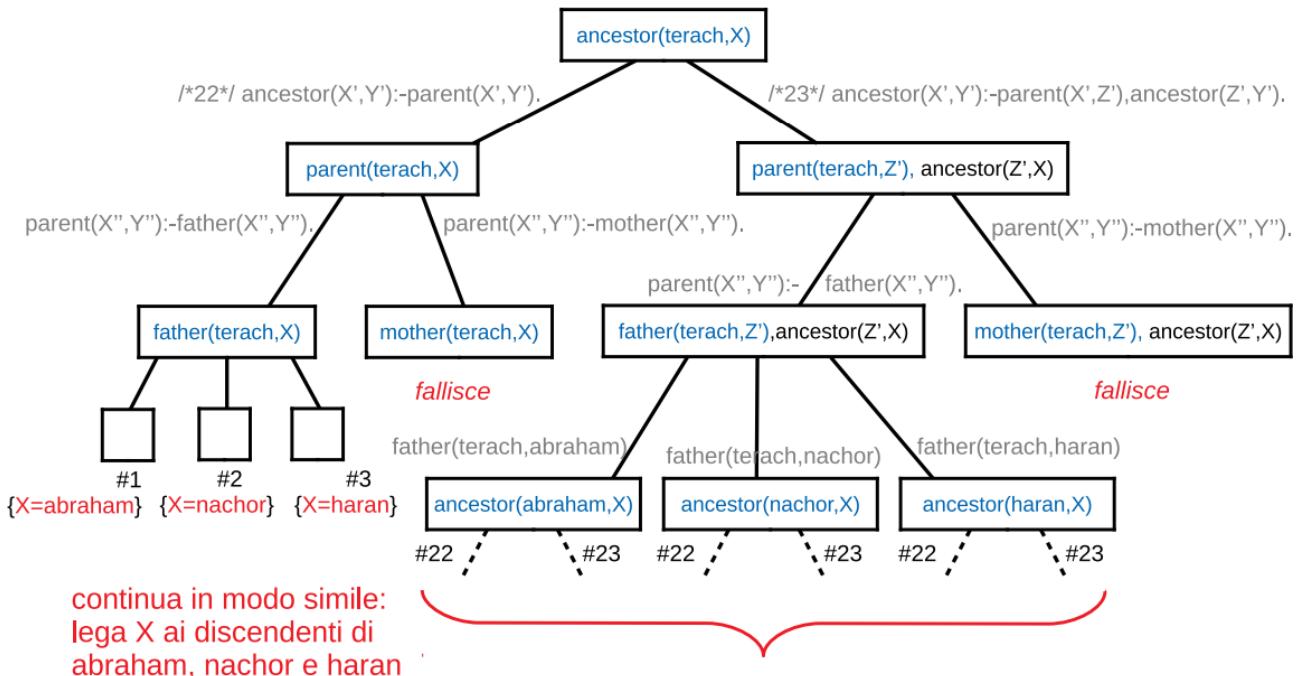
Un altro esempio di regola ricorsiva la si può fare con il programma sulla genealogia biblica, aggiungendo la nozione di antenato (ancestor). Definizione ricorsiva:  $X$  è un antenato di  $Y$  se vale una di queste condizioni:

1.  $X$  è genitore di  $Y$  (caso base)
2.  $X$  è genitore di  $Z$  e  $Z$  è antenato di  $Y$  (per qualche  $Z$ )

```

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

```



## Prolog e l'algebra relazionale

Ogni tabella relazionale  $r$  si può rappresentare con dei fatti:

| $a_1$    | $b_1$    | $c_1$    | $d_1$    |
|----------|----------|----------|----------|
| $a_2$    | $b_2$    | $c_2$    | $d_2$    |
| $a_3$    | $b_3$    | $c_3$    | $d_3$    |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |

```

r(a1, b1, c1, d1).
r(a2, b2, c2, d2).
r(a3, b3, c3, d3).
.
.
.

```

e con le regole si può facilmente simulare l'algebra relazionale (ovvero le query SQL). Prolog genera le n-uple della risposta una per una:

```
/* UNIONE di r e s */
r_union_s(X1,...,Xn) :- r(X1,...,Xn).
r_union_s(X1,...,Xn) :- s(X1,...,Xn).

/* INTERSEZIONE di r e s */
r_inters_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).

/* PROIEZIONE di r, ad esempio su colonne 1 e 3 */
r13(X1,X3) :- r(X1,...,Xn), <condizione>.

% ad esempio
r_with_2_less_than_3(X1,...,Xn) :- r(X1,...,Xn), X2 < X3.
```

Per esercizio si scriva il prodotto cartesiano e join su colonne 1 e 2.

Il Prolog permette inoltre, di poter descrivere l' implicazioni:  $(A \leftarrow B) \wedge (A \leftarrow C) \equiv A \leftarrow (B \vee C)$  scrivendo semplicemente  $A : -B ; C$ . Il ; sostituisce l'OR logico ed ha una precedenza più bassa dell'AND, quindi  $A : -B , C ; D$  equivale a scrivere  $A : -(B, C) ; D$

Per la differenza tra relazioni occorre un operatore di negazione che in prolog si denota con  $\backslash+$  ( oppure con la parola chiave *not*).

```
/* DIFFERENZA di r e s */
r_minus_s(X1,...,Xn) :- r(X1,...,Xn), \+ s(X1,...,Xn).
```

La **negazione** trasforma false in true, ovvero fallimenti in successi.

- $\backslash+ p(X_1, \dots, X_n)$  è true se tutti i rami del suo albero di ricerca terminano con un fallimento
- L'albero deve essere finito. Se un programma cade in una ricorsione infinita ovviamente non termina.

## 69. Liste

### Sintassi

La rappresentazione è superficialmente analoga a quella in ML, quindi avremo un costruttore di lista vuota: `[]`; ed un costruttore di nodi: `[elem|resto]` (differisce da ML, dove si può definire con `elem :: resto`).

Di seguito si riportano le notazioni alternative equivalenti:

| Abbreviata             | Costruttori esplicativi     |
|------------------------|-----------------------------|
| <code>[a]</code>       | <code>[a []]</code>         |
| <code>[a,b]</code>     | <code>[a [b []]]</code>     |
| <code>[a,b,c]</code>   | <code>[a [b [c []]]]</code> |
| <code>[a X]</code>     | <code>[a X]</code>          |
| <code>[a,b X]</code>   | <code>[a [b X]]</code>      |
| <code>[a,b,c X]</code> | <code>[a [b [c X]]]</code>  |

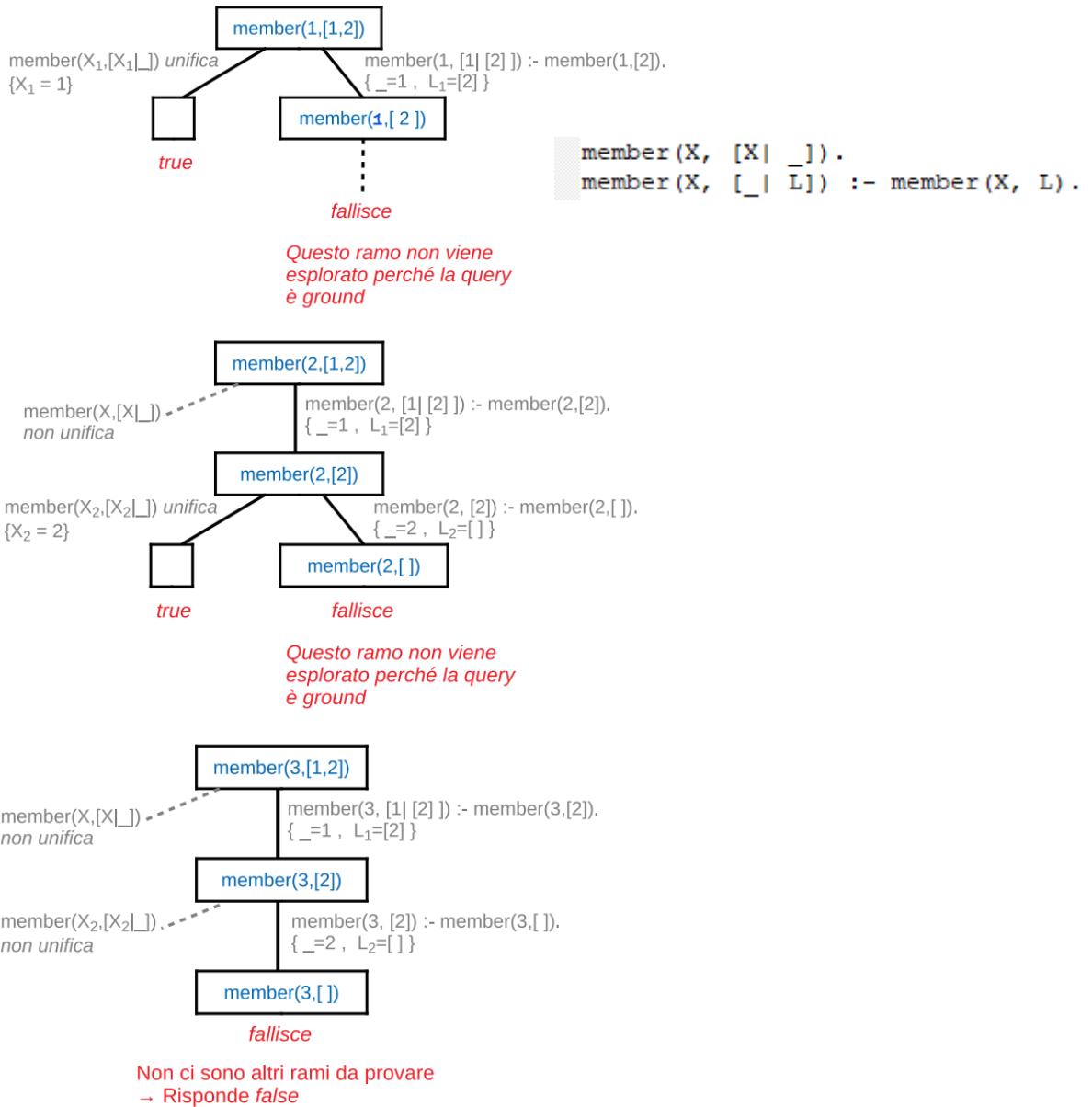
Notare la possibilità di esprimere liste parzialmente specificate, dove alcuni elementi ed eventualmente la coda sono variabili: `[a, X, b|Y]`

### Il predicato member

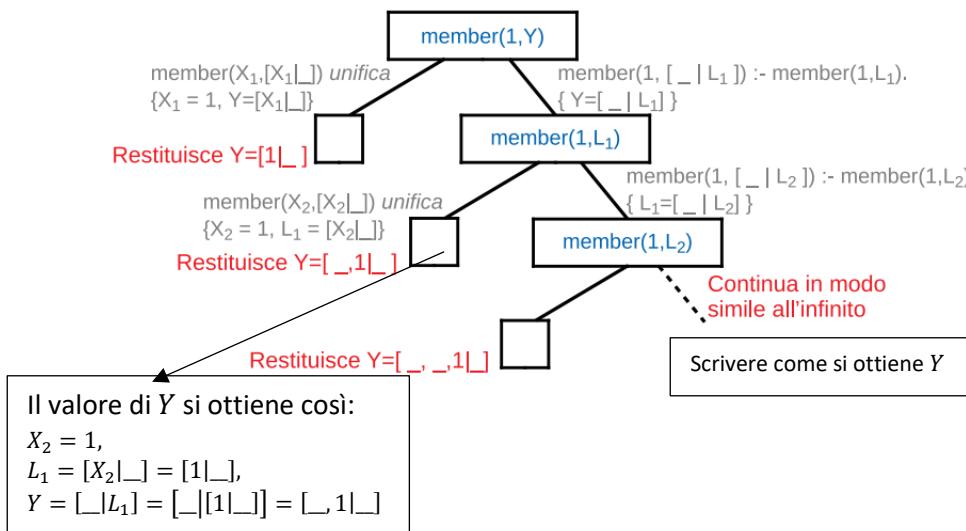
Il predicato member cerca un elemento  $X$  in una lista  $L$ , ovviamente, ricorsivamente:

- Caso base:  $member(X, [X|_])$ . per ogni  $X$ ,  $X$  è membro di qualunque lista se compare come primo elemento.
- Caso ricorsivo:  $member(X, [_|L]) :- \neg member(X, L)$ .  $X$  è membro di una lista se  $X$  è membro di tutto quello che viene dopo il primo elemento.

Somiglia alla definizione per casi in ML ma in Prolog posso usare la stessa variabile più volte per esprimere pattern dove certi elementi sono uguali. Di seguito riportiamo degli esempi di derivazione per member:



Notare che mentre qui risponde false in ML avrebbe sollevato una eccezione.



## Evanescenza di parametri di Input e Output

In Prolog non c'è una chiara distinzione tra parametri IN e OUT, infatti ogni parametro può essere legato a un termine con costruttori (input) ed ogni parametro attuale con una variabile libera produce delle sostituzioni (output), mentre i parametri con almeno un costruttore e una variabile forniscono sia un input sia un output.

Guardiamo gli esempi con member:

- $member(X, \langle lista\ ground \rangle)$  prende una lista e restituisce i suoi elementi. Modalità: (OUT,IN)
- $member(\langle elem.\ ground \rangle, L)$  prende un elemento e restituisce le liste che lo contengono. (IN,OUT)
- **Invertibilità dei prediciati:** possono rappresentare sia una funzione sia la sua inversa

Incontreremo lo stesso fenomeno nel predicato append con 3 argomenti.

### Il predicato append

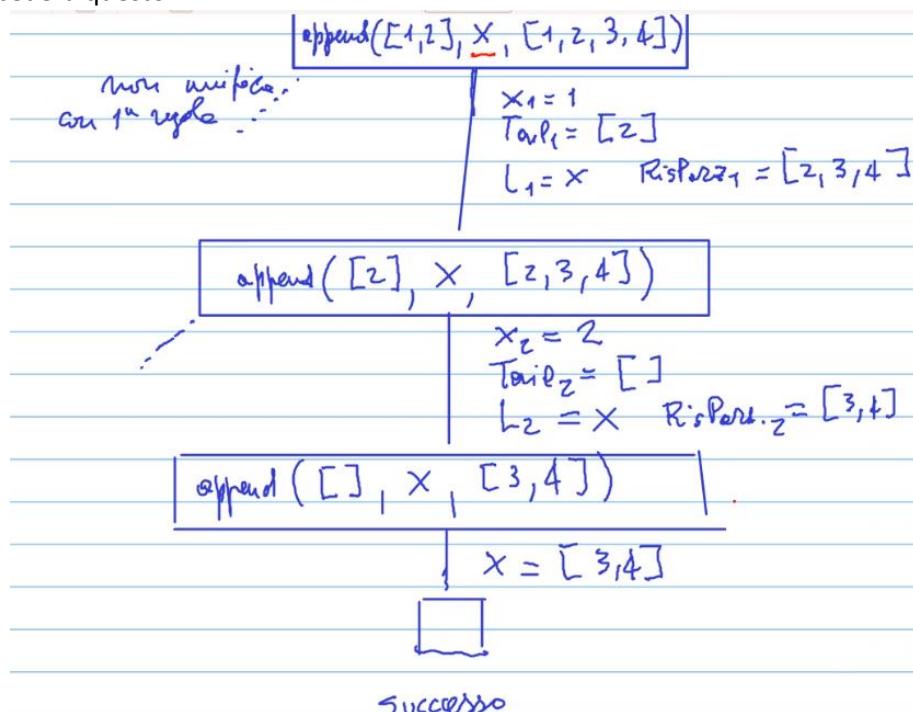
Prima di proseguire si svolgano i seguenti esercizi:

- 1) Scrivere un predicato append che concatena due liste. Schema:  $append(Lista1, Lista2, Risultato)$
- 2) Usare l'invertibilità di append per verificare se  $[a, b, c]$  è un prefisso di una lista data.
  - verificare disegnando il search tree per la lista  $[a, b, c, d]$
- 3) Usare l'invertibilità di append per verificare se  $[a, b, c]$  è un suffisso di una lista data.
  - verificare disegnando il search tree per la lista  $[a, a, b, c]$
- 4) Usare l'invertibilità di append per definire un predicato  $last(L, X)$  che è vero se  $X$  è l'ultimo elemento della lista  $L$ .

Per fare l'append di due liste serve un'implementazione ricorsiva sulla prima lista, quindi il caso base è quando la lista è vuota, ciò significa che non ci interessa la prima lista, di conseguenza il nostro caso base sarà  $append([], L, L)$ , a questo punto bisogna fare la chiamata ricorsiva per concatenare  $Tail$  con  $L$  e restituiamo il risultato parziale con in testa la  $X$ :  $append([X|Tail], L, [X|RisParx])$  :

–  $append(Tail, L, RisParx)$ . Dove ad esempio la concatenazione di  $[1,2]$  e  $[3,4,5]$  è uguale a  $[1|[2,3,4,5]]$ .

Se volessi ad esempio cercare la  $X$  in  $append([1,2], X, [1,2,3,4])$  esso mi darà come risultato  $[3,4]$ , nell'albero succederà questo:



Di seguito riportiamo alcuni risultati di query:

- Chiediamo di avere solo il prefisso:

```
?- append(X, [3,4,5], [1,2,3,4,5]).
X = [1,2];
false. % se chiedo un' altra soluzione.
```

- Chiediamo soluzioni sia per il prefisso che per il suffisso fino ad avere tutte le soluzioni possibili:

```
?- append(Pre, Suf, [1,2,3,4,5]).
Pre = [],
Suf = [1, 2, 3, 4, 5];
Pre = [1],
Suf = [2, 3, 4, 5];
Pre = [1, 2],
Suf = [3, 4, 5];
Pre = [1, 2, 3],
Suf = [4, 5];
Pre = [1, 2, 3, 4],
Suf = [5];
Pre = [1, 2, 3, 4, 5],
Suf = [];
false.
```

- Posso addirittura avere elementi liberi (quindi potenzialmente risultati infiniti):

```
?- append(X, [2,3], L).
X = [],
L = [2, 3];
X = [_2626],
L = [_2626, 2, 3];
X = [_2626, _2638],
L = [_2626, _2638, 2, 3];
...
...
```

- Oppure avere tutte le variabili come incognite:

```
?- append(X, Y, L).
X = [],
Y = L;
X = [_2342],
L = [_2342|Y];
X = [_2342, _2354],
L = [_2342, _2354|Y];
...
...
```

- O usarlo come una funzione booleana:

```
?- append([1], [2], [1,2]).
true.

?- append([], [2], [1,2]).
false.
```

Grazie ad append potrei creare una regola che generi tutti i prefissi/suffissi di una lista:

```
prefix(Prefix, List) :- append(Prefix, _, List).
suffix(Suffix, List) :- append(_, Suffix, List).
```

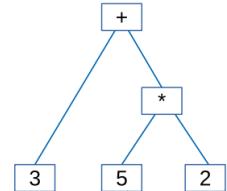
Sempre sfruttando la funzione append potrei definire una funzione member che mi dice quando un elemento compare all'interno di una lista:  $member(X, L) :- append([_ | L], L)$ .

Possiamo anche avere il calcolo delle sottoliste (intersezione tra prefisso e suffisso) di  $L$ . Notare che  $S$  è una sottolista di  $L$  se vale la seguente condizione:  $S$  prefix di un suffix di  $L \Leftrightarrow S$  suffix di un prefix di  $L$

## 70. Applicazioni

### Calcolo simbolico in Prolog

In Prolog gli operatori aritmetici sono costruttori, infatti se scrivo  $3 + 5 * 2$  questo non è uguale a 13 ma denoterà l'albero sintattico a destra, esso quindi non valuta l'espressione ma la usa come una query. Per calcolare l'operazione si usa la parola chiave *is*, infatti il goal  $R \text{ is } 3 + 5 * 2$  è vero se  $R$  è uguale al valore di  $3 + 5 * 2$ . Invece, il simbolo  $=$  serve ad unificare i risultati. Esempi di query sono:



```
?- R is 3+5*2. ?- R = 3+5*2. ?- 13 is 3+5*2. ?- 12 is 3+5*2.
R = 13. R = 3+5*2. true. false.
```

Il fatto che gli operatori aritmetici vengano considerati come costruttori rendono il calcolo simbolico particolarmente semplice. Facciamo un esempio basato sulle derivate, stile Matlab.

### Calcolo simbolico delle derivate

Consideriamo il seguente schema del predicato:  $der(Exp, X, D)$  è vero se  $D$  è la derivata di  $Exp$  rispetto a  $X$ . E quindi il nostro programma sarà:

```
der(X, X, 1).
der(X^N, X, N*X^(N-1)) :- N1 is N-1.
der(sen(X), X, cos(X)).
der(cos(X), X, -sen(X)).
der(log(X), X, 1/X).
der(F+G, X, DF+DG) :- der(F, X, DF), der(G, X, DG).
der(F-G, X, DF-DG) :- der(F, X, DF), der(G, X, DG).
der(F*G, X, F*DG + DF*G) :- der(F, X, DF), der(G, X, DG).
```

Un esempio di query sarà dunque:

```
?- der(x^3 * cos(x), x, D).
D = x^3 * -sen(x) + 3 * x^2 * cos(x).
```

[si possono aggiungere predicati per semplificare il risultato]

## 71. Programmazione nondeterministica

### In che consiste

Invece di dire a Prolog *come* trovare la soluzione di un problema si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree). Lo schema generale (generate and test) è il seguente:  $solution(X) :- generate(X), test(X)$ .

- Prolog genera via via i candidati a soluzione  $X$
- Per ciascuno di essi verifica se è effettivamente una soluzione (test)
- Se sì, restituisce la soluzione; altrimenti fa backtrack e torna a generare il successivo candidato
- Se si chiedono altre risposte, genera altre soluzioni.

La programmazione nondeterministica è una caratteristica unica del paradigma logico, infatti negli altri paradigmi la si può solo approssimare:

```
/* paradigma imperativo */
X := primo candidato;
while not test(X) and X != null do
 X := prossimo_candidato(X)
return X
```

```
/* paradigma funzionale */
fun soluzione(X) =
 if test(X) then X
 else soluzione(prossimo_candidato(X))
/* invocare così */
soluzione (primo_candidato);
```

In entrambi i casi verrebbe generata solo la prima soluzione trovata (se si vogliono le altre occorre complicare il codice). L'utilità del generare tutte le soluzioni sarà illustrata programmando l'AI di un gioco.

Vediamo, per dare un primo esempio di programmazione nondeterministica una ricerca dei membri pari di una lista:

```
/* stile generate and test */
membro_pari(X, L) :- member(X, L), 0 is X mod 2.

/* invece di ricorsione ad hoc */
membro_pari(X, [X|_]) :- 0 is X mod 2.
membro_pari(X, [_Resto]) :- membro_pari(X, Resto).
```

Notare come l'approccio generate and test possa giocare il ruolo delle funzioni di ordine superiore in ML, permette, infatti, di comporre facilmente nuovi predicati da quelli dati ed in questo caso *member*, che diventa uno strumento generale per visitare una lista funge da filter: la query congiuntiva *member (X ,Lista ) ,predicato(X )*. è l'analogo di *filter predicato Lista*.

## Applicazione ai giochi

Un possibile pattern generate and test per i giochi è il seguente:

```
next_move(Player, Move) :-
 possible_move(Player, Move), optimal(Player, Move).
```

A sua volta il controllo di ottimalità (cioè verificare se con *Move* sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test.

Nota: *optimal* mi dice se *Move* è la prima mossa di una strategia di gioco; ad esempio “il primo giocatore ha una strategia vincente?” mostrerà che è utile generare tutte le soluzioni possibili.

Descriviamo un semplice programma (imbattibile) di AI che gioca a tris formato, grazie al paradigma logico, da una sola pagina di codice per “l'intelligenza” e da una pagina per i turni e l'interfaccia utente. Il nostro programma effettuerà ad ogni mossa una ricerca della strategia ottima, ovvero, se esiste una strategia vincente la adotta, altrimenti mira al pareggio.

Approccio generate and test (programmazione nondeterministica) alla scelta della mossa.

Ad ogni turno:

1. Genera una mossa
2. Verifica se è ottimale (la prima di una strategia ottima)

### Descrizione del gioco:

```
% rappresentazione scacchiera
start([[1, 2, 3], [4, 5, 6], [7, 8, 9]]).

adversary (x, o).
adversary (o, x).
```

```

% il Player vince se occupa le seguenti posizioni
win(P, [[P, P, P], _, _]).
win(P, [_, [P, P, P], _]).
win(P, [_, _, [P, P, P]]).
win(P, [[P, _, _], [P, _, _], [P, _, _]]).
win(P, [[_, P, _], [_, P, _], [_, P, _]]).
win(P, [[_, _, P], [_, _, P], [_, _, P]]).
win(P, [[P, _, _], [_, P, _], [_, _, P]]).
win(P, [[_, _, P], [_, P, _], [P, _, _]]).

% stato in cui il gioco non termina
non_final(Board) :-
 \+ win(_, Board),
 member(Row, Board),
 member(Cell, Row),
 number(Cell).

% stato in cui il gioco termina
final(Board) :-
 \+ non_final(Board).

```

### Possibili mosse e loro effetto

Schema: `move(+P,N,+Board1,-Board2)` dove:

- $P$  (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- $N$  la mossa, indicata dal numero della cella ( $1 \leq N \leq 9$ )
- $Board1$  è l'attuale scacchiera
- $Board2$  è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog:  $+ = IN, - = OUT, niente = IN/OUT$

```

move(P, N, Board1, Board2) :-
 \+ win(_, Board1),
 append(RowsBefore, [Row|RowsAfter], Board1),
 append(CellsBefore, [N|CellsAfter], Row),
 number(N),
 append(CellsBefore, [P|CellsAfter], NewRow),
 append(RowsBefore, [NewRow|RowsAfter], Board2).

```

### Le strategie (generate and test)

Schema: `has_XXX_strat(+P,-Move,+Board)` dove:

- $P$  (player) è il simbolo 'x' oppure 'o'
- $Move$  è la prima mossa della strategia
- $Board$  è l'attuale scacchiera

Significato: nella situazione descritta da  $Board$ ,  $P$  ha una strategia di tipo XXX (vincente o non-perdente) che inizia con  $Move$

```

has_win_strat(P, _, Board) :-
 win(P, Board).

has_win_strat(P, Move, Board) :-
 move(P, Move, Board, Board2),
 adversary(P, Adv),
 \+ has_tie_strat(Adv, _, Board2).

```

```

has_tie_strat(_, _, Board) :-
 final(Board),
 \+ win(_, Board).

has_tie_strat(P, Move, Board) :-
 move(P, Move, Board, Board2),
 adversary(P, Adv),
 \+ has_win_strat(Adv, _, Board2).

```

### L'interfaccia utente testuale

```
print_board([]).
print_board([Row|Rest]) :-
 format('~a|~a|~a \n', Row),
 print_board(Rest).

read_move(Player, Move) :-
 format('\nPlayer ~a insert your move [1-9]: ', [Player]),
 get_single_char(Char), put_char(Char), nl,
 Move is Char-48, % sottrae il codice ascii dello 0
 Move >= 1, Move =< 9.
```

```
cpu move: 1
x|2|3
4|5|6
7|8|9

Player o insert your move [1-9]: 8
x|2|3
4|5|6
7|o|9
```

### I turni

```
turn(_, _, Board) :-
 final(Board),
 \+ win(_, Board),
 format('\nThe game ends in a tie. \n').

turn(P, _, Board) :-
 win(_, Board),
 member(Adv, [user, cpu]), Adv \= P,
 format('\nThe ~a wins! \n', [Adv]).

turn(user, P, Board) :-
 read_move(P, M),
 move(P, M, Board, Board2),
 print_board(Board2),
 adversary(P, Adv),
 turn(cpu, Adv, Board2).

turn(cpu, P, Board) :-
 (has_win_strat(P, Move, Board); has_tie_strat(P, Move, Board)),
 format('\ncpu move: ~a \n', [Move]),
 move(P, Move, Board, Board2),
 print_board(Board2),
 adversary(P, Adv),
 turn(user, Adv, Board2).
```

### Il main

Due modalità di esecuzione:

```
% Specifica tramite query chi inizia e con che simbolo
go(First, Symbol) :-
 member(First, [user, cpu]), % Esempio di invocazione
 start(Board), % ?- go(user, x).
 turn(First, Symbol, Board).

% Estrazione a sorte di chi inizia
main : -
 Choice is random(2),
 nth0(Choice, [(user, o), (cpu, x)], (First, Symbol)),
 format('\nThe ~a starts \n', [First]),
 start(Board),
 turn(First, Symbol, Board),
 halt.
```

## Analisi del gioco

Non esiste nessuna mossa iniziale che garantisca al primo giocatore di vincere, infatti:

```
?- start(Board), has_win_strat(x, Move, Board).
false.
```

Non esiste nemmeno una strategia vincente per il secondo giocatore, infatti ogni mossa iniziale permette al primo giocatore di pareggiare (se non commette errori) e dunque tutte le mosse iniziali (per il secondo giocatore) fanno parte di una strategia di pareggio:

```
? - start(B0), has_tie_strat(x, Mv, B0).
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
Mv = 1;
...
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
Mv = 9;
false.
```

Come può pareggiare il secondo giocatore?

```
?- start(B0), move(x, 1, B0, B1), has_tie_strat(o, Mv, B1).
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
B1 = [[x, 2, 3], [4, 5, 6], [7, 8, 9]],
Mv = 5;
false.
```

- Se la prima mossa è su un angolo, l'unica risposta è 5, in tutti gli altri casi vince il primo giocatore

```
?- start(B0), move(x, 2, B0, B1), has_tie_strat(o, Mv, B1).
Mv = 1;
Mv = 3;
Mv = 5;
Mv = 8;
false.
```

- Quindi se la prima mossa è 2, meglio non rispondere con 4, 6, 7 o 9!

```
?- start(B0), move(x, 5, B0, B1), has_tie_strat(o, Mv, B1).
Mv = 1;
Mv = 3;
Mv = 7;
Mv = 9;
false.
```

- Se la prima mossa è 5, si deve scegliere una casella d'angolo

Tutte queste affermazioni si possono verificare empiricamente forzando il primo passo:

```
?- start(B0), move(x, 5, B0, B1), turn(user, o, B1).
```

## Compilazione in codice stand-alone

```
swipl --goal=main --stand_alone=true -o nomefile -c nomefile.pl
```

- *--stand\_alone*: crea un codice direttamente eseguibile (invece di far partire la macchina virtuale)
- *-o*: nome del file oggetto
- *-c*: nome del file sorgente
- *--goal*: il goal da invocare quando il file viene eseguito

Riassumendo: caratteristiche uniche di Prolog

**Invertibilità dei predicati:** un singolo predicato implementa molte funzioni; dovuto al fatto che le variabili logiche sono IN/OUT/IN-OUT a seconda dei parametri attuali.

**Programmazione nondeterministica:** con ricerca automatica delle soluzioni; basato sul backtracking (cioè il meccanismo di visita dei search tree).

Ovviamente ci sono molti altri aspetti interessanti come strutture dati parziali (permettono append e creazione dizionari in tempo costante ), meta-predicati/reflection, aggregati (setof), forall, etc...