

# 1 - Interpreti, Compilatori, Macchine Astratte

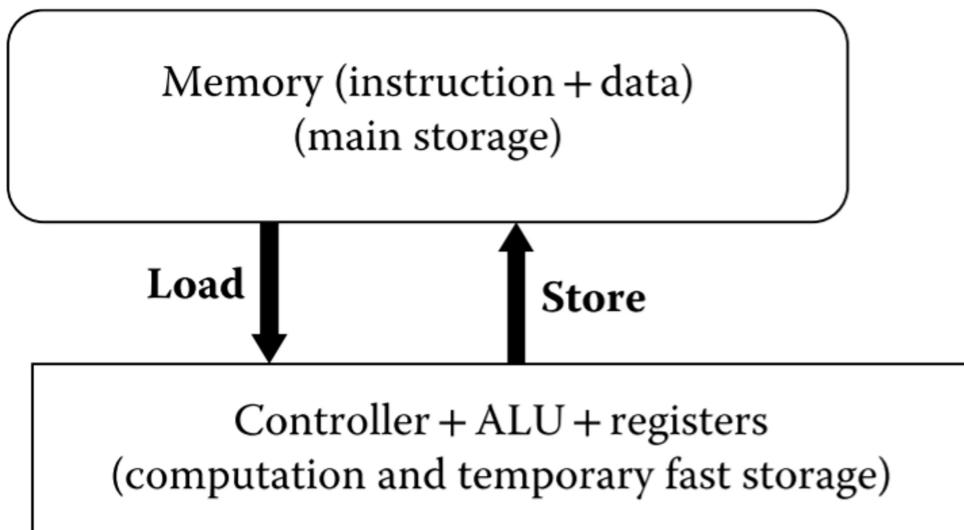
giovedì 23 dicembre 2021 11:10

## Macchina di von neumann

Base della struttura dei computer attuali.

Componenti principali:

- Memoria
- Unità centrale di elaborazione: ha il compito di eseguire i programmi immagazzinati in memoria prelevando le istruzioni (e i dati, dalla memoria) interpretandole e eseguendole in sequenza



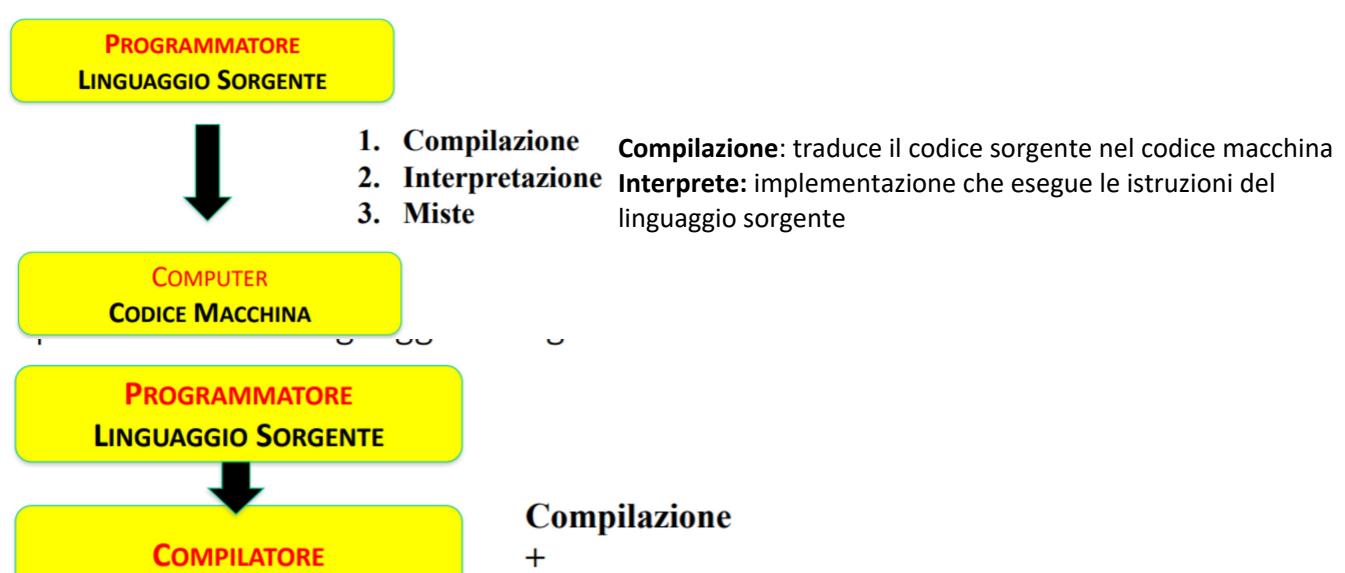
## Ciclo fetch-execute

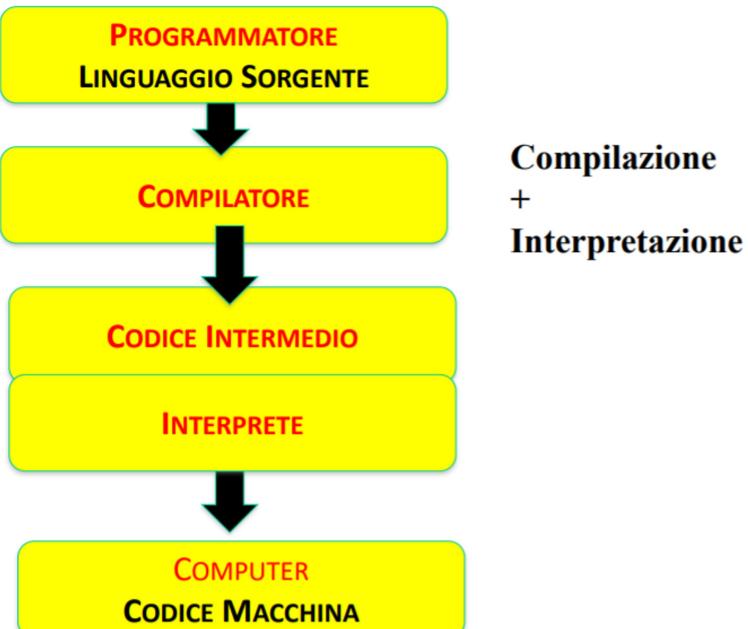
- Fetch: istruzione prelevata dalla memoria e trasferita all'interno della cpu
- Decode: istruzione viene interpretata e viene preparata la sua esecuzione
- Data fetch: prelevati i dati sui quali eseguire l'operazione prevista dall'istruzione
- Execute: viene eseguita l'operazione
- Store: il risultato dell'operazione viene memorizzato

**Architetture "tradizionali":** il ciclo viene eseguito in modo sequenziale, ogni istruzione viene completata prima dell'avvio di quella successiva

**Architetture "moderne":** l'istruzione successiva inizia ad essere elaborata prima che venga completata la precedente il ciclo è suddiviso in fasi separate)

## IMPLEMENTAZIONE LINGUAGGI DI PROGRAMMAZIONE





### Macchine astratte:

Una macchina stratta è un sistema virtuale che rappresenta il comportamento di una macchina fisica, individuando:

- L'insieme delle risorse necessarie per l'esecuzione di programmi
- Un insieme di istruzioni specificatamente progettato per operare con queste risorse

### Macchine astratte per linguaggi funzionali:

Stack: gestire le chiamate di funzione

Environment: gestire associazioni variabile-valore

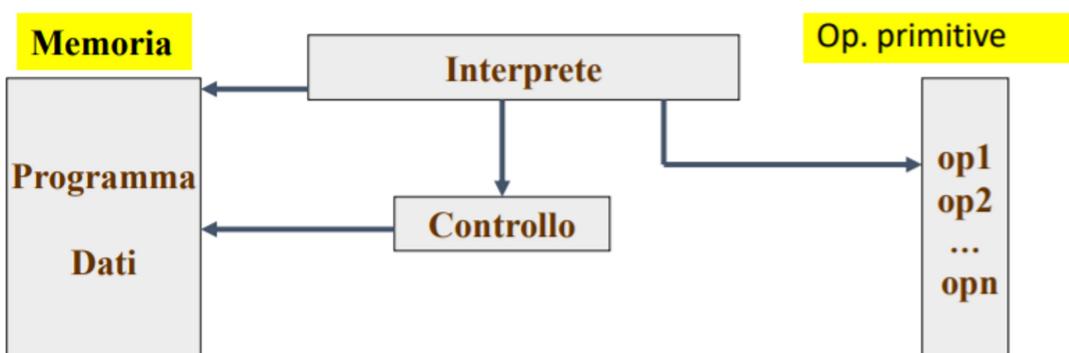
Closure: rappresentare il valore di una funzione

Heap&Garbage collector: gestione della memoria dinamica

Una collezione di strutture dati e algoritmi in grado di memorizzare ed eseguire programmi

Componenti:

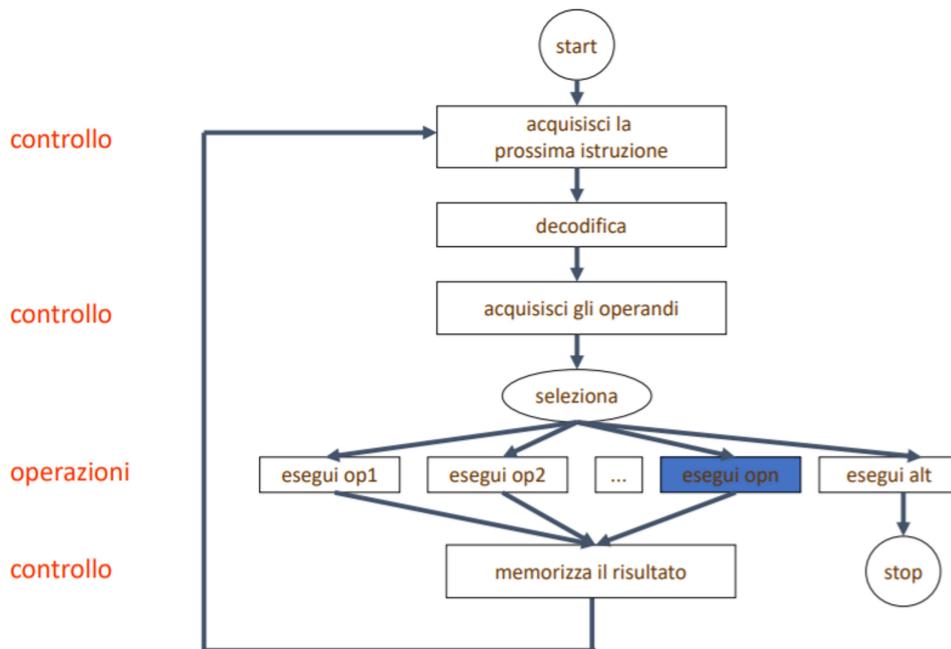
- Interprete
- Memoria
- Controllo
- Operazioni "primitive"



33

**Componente di controllo:** collezione di strutture dati e algoritmi per eseguire le varie operazioni necessarie a eseguire un programma (acquisire istruzioni, acquisire operandi, memorizzare risultati delle operazioni, gestire chiamate di funzione.)

### INTERPRETE



L'interprete è un programma che prende in ingresso il programma da eseguire (l'albero di sintassi astratta) e lo esegue ispezionando la struttura del programma per vedere cosa deve essere fatto

### Macchine astratte e linguaggio macchina:

M: macchina astratta

L<sub>m</sub>: linguaggio macchina di M (linguaggio che ha come stringhe legali tutti i programmi interpretabili dall'interprete di M)

Alle componenti di M corrispondono componenti di L<sub>m</sub>

- Tipi di dato primitivi
- Costrutti di controllo
  - o Controllare l'ordine di esecuzione
  - o Controllare l'esecuzione e trasferimento dati

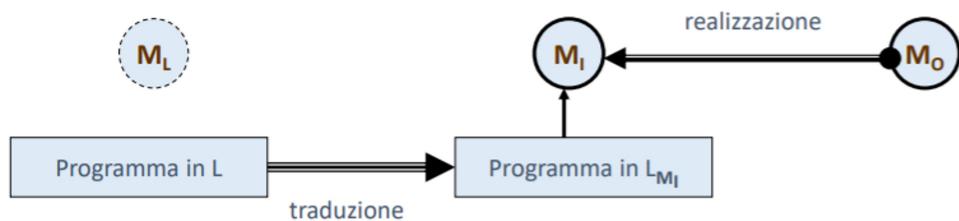
- **M** macchina astratta
- I componenti di **M** sono realizzati mediante strutture dati e algoritmi implementati nel linguaggio macchina di una **macchina ospite M<sub>o</sub>**, già esistente (implementata)
- È importante la realizzazione dell'interprete di **M**
  - o può coincidere con l'interprete di **M<sub>o</sub>**
    - ✓ **M** è realizzata come **estensione** di **M<sub>o</sub>**
    - ✓ altri componenti della macchina possono essere diversi
  - o può essere diverso dall'interprete di **M<sub>o</sub>**
    - ✓ **M** è realizzata su **M<sub>o</sub>** in modo **interpretativo**
    - ✓ altri componenti della macchina possono essere uguali

Implementare un linguaggio:

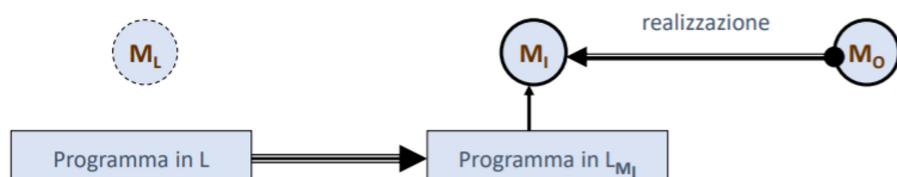
- **M** macchina astratta      L<sub>M</sub> linguaggio macchina di **M**
- **L** linguaggio      M<sub>L</sub> macchina astratta di **L**
- **Implementazione di L** =
  - realizzazione di M<sub>L</sub> su una macchina ospite M<sub>o</sub>

- $L$  linguaggio ad alto livello
- $M_L$  macchina astratta di  $L$
- $M_O$  macchina ospite
- **interprete (puro)**
  - $M_L$  è realizzata su  $M_O$  in modo interpretativo
  - scarsa efficienza, soprattutto per colpa dell'interprete (ciclo di decodifica)
- **compilatore (puro)**
  - i programmi di  $L$  sono tradotti in **programmi funzionalmente equivalenti** nel linguaggio macchina di  $M_O$
  - i programmi tradotti sono eseguiti direttamente su  $M_O$
  - $M_L$  non viene realizzata
  - il problema è quello della dimensione del codice prodotto
- Casi limite che nella realtà non esistono quasi mai

Macchina intermedia



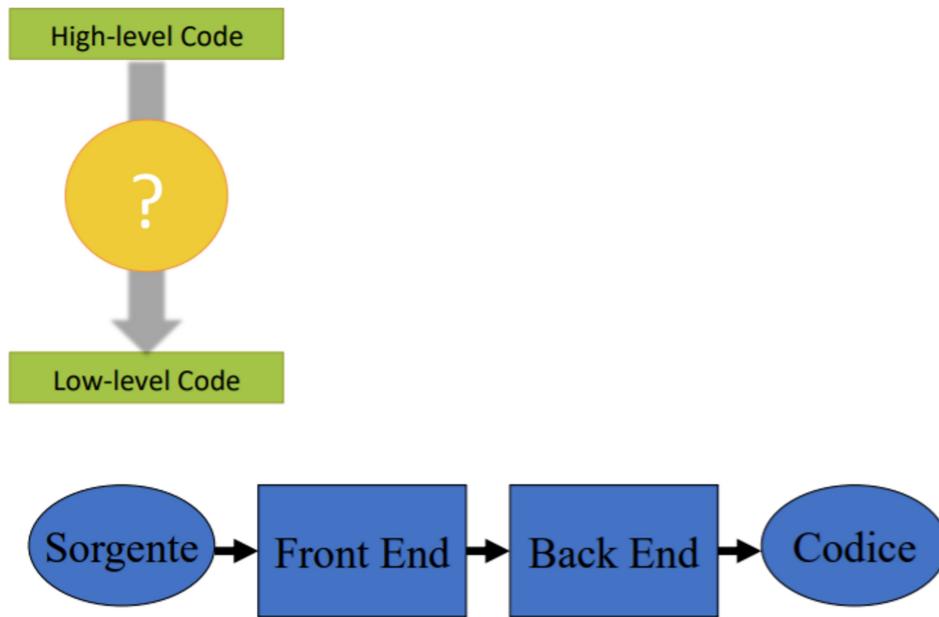
- 
- $L$  linguaggio ad alto livello
  - $M_L$  macchina astratta di  $L$
  - $M_I$  macchina intermedia
  - $L_{M_I}$  linguaggio intermedio
  - $M_O$  macchina ospite
    - traduzione dei programmi da  $L$  al linguaggio intermedio  $L_{M_I}$
    - realizzazione della macchina intermedia  $M_I$  su  $M_O$



- 
- $M_L = M_I$  interpretazione pura
  - $M_O = M_I$  traduzione pura
    - possibile solo se la differenza fra  $M_O$  e  $M_L$  è molto limitata
      - $L$  linguaggio assembler di  $M_O$
    - in tutti gli altri casi, c'è sempre una macchina intermedia che estende eventualmente la macchina ospite in alcuni componenti

Qui penso sia buono guardare il pezzo di lezione in cui lo spiega

## COMPILATORE



Front end: analisi:

Legge il programma sorgente e determina la sua struttura, sia sintattica che semantica

Back end: sintesi:

Genera il codice nel linguaggio macchina, programma equivalente al programma sorgente

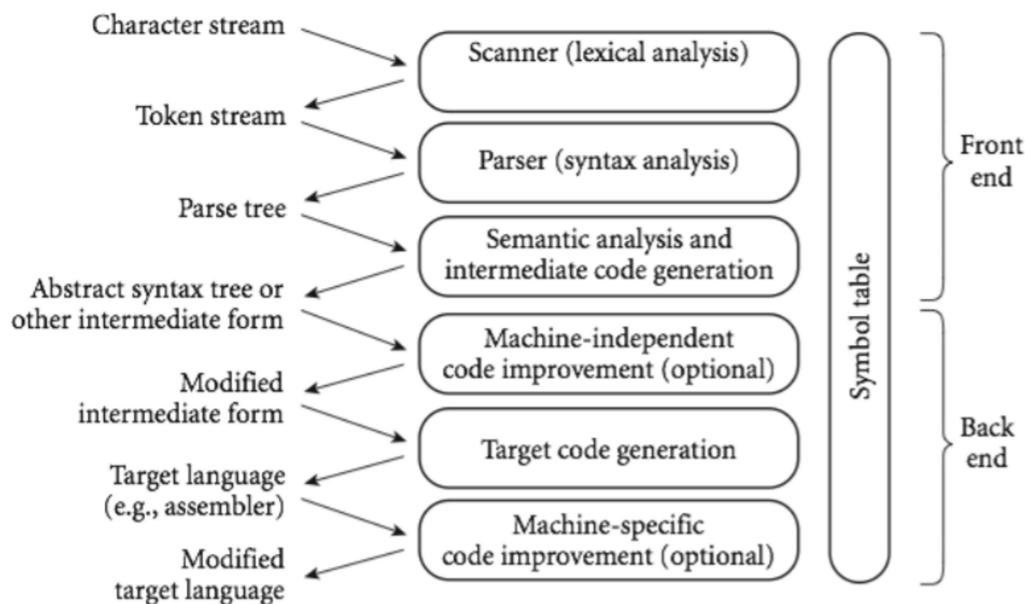
Aspetti critici:

Riconoscere programmi sintatticamente corretti

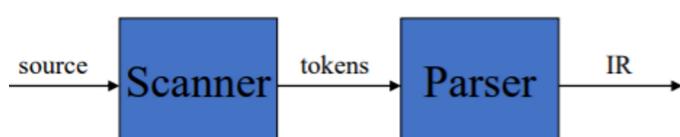
Gestire i tipi

Generare codice compatibile col SO della macchina ospite

### Struttura di un compilatore



### Front end:



Scanner: trasforma il programma sorgente nel lessico (token)

Parser: legge i token e genera il codice intermedio (IR)

*Token*: costituente lessicale del linguaggio (operatori, parole chiave, identificatori, costanti...)

Es:

`if (x >= y) y = 42;`

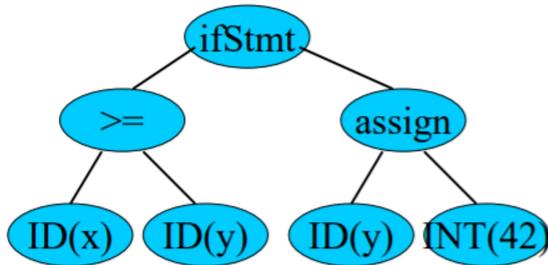
Token:

IF	LPAREN	ID(x)	GEQ	ID(y)
RPAREN	ID(y)	BECOMES	INT(42)	SCOLON

Parser:

Output:

- Formati differenti
- Formato tipico riconosciuto: albero di sintassi astratta



IF	LPAREN	ID(x)	GEQ	ID(y)
RPAREN	ID(y)	BECOMES	INT(42)	SCOLON

**AST:**

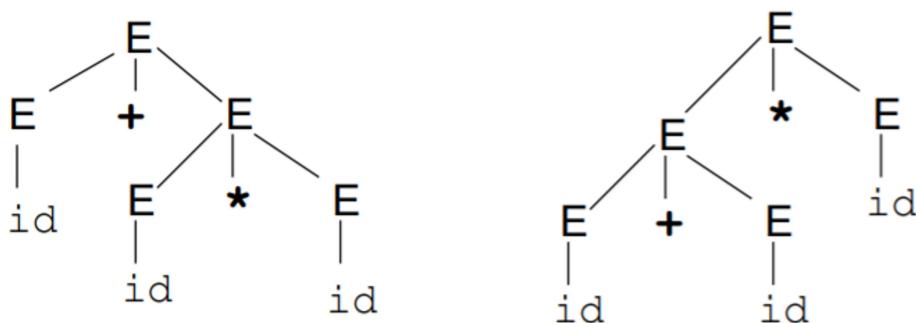
Gli alberi di sintassi astratta mostrano la struttura semantica significativa dei programmi

**Ambiguità:**

Programma corretto con AST diversi

Esempio:

$E ::= E+E \mid E^*E \mid id$



Si risolve ad esempio codificando nelle regole della grammatica la precedenza degli operatori:

$E \rightarrow E+E \mid E^*E$

$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$

La teoria aiuta a strutturare le grammatiche in modo tale da evitare problemi del genere

**Parser a discesa ricorsiva**

Parser che parte dal simbolo iniziale della grammatica e applicando le regole (produzioni della grammatica) cerca di riscrivere il simbolo iniziale fino a ottenere il programma in ingresso

Implementazione: insieme di funzioni mutualmente ricorsive in cui ciascuna implementa una delle regole della grammatica

La struttura di un parser a discesa ricorsiva è derivata e rispecchia la struttura della grammatica che riconosce

### Sintassi astratta

Espressa facilmente coi tipi di dato algebrici di Ocaml: ogni categoria sintattica diventa un tipo di dato algebrico di ocaml

### BNF

BoolExp =

- | True
- | False
- | NOT BoolExp
- | BoolExp AND BoolExp

### Algebraic Data Type

Type BoolExp =

- | True
- | False
- | Not of BoolExp
- | And of BoolExp \* BoolExp

Dopo la fase di parsing:

- Type checking
- Uso e allocazione delle risorse
- Varie analisi statiche
- Ottimizzazione del codice

### Analisi statica: esempi

(controlli sull'inizializzazione delle variabili)

### control flow analysis

```
if (b) { c = 5; } else { c = 6; } initialises c
if (b) { c = 5; } else { d = 6; } does not
```

### data flow analysis

d = 5; c = d;	<b>initialises c</b>
c = d; d = 5;	<b>does not</b>

## Analisi statica: esempi (identificazione del codice inutilizzato)

```
void foo()
{
    int x;

    x = read_password();
    use(x);
    x = 0; // clear password
    untrusted();
    return;
}
```

## Analisis statica: esempi (identificazione del codice inutilizzato)

```
void foo()
{
    int x;

    x = read_password();
    use(x);
    x = 0; // clear password
    untrusted();
    return;
}
```

Poiché il valore 0 memorizzato in x  
non viene mai utilizzato,  
L'assegnamento (x = 0) viene  
rimosso (silenziosamente)  
dall'ottimizzazione statica  
dead-code elimination

### Taint analysis

Tracciare come le informazioni fluiscano attraverso un programma staticamente

1. Identificare le fonti di contaminazione
2. Identificare la propagazione di contaminazione

X = get\_input(...) x è tainted

Input t = IsUntrusted(src)  
get\_input(src): tainted

Y = x + 42 -> y è tainted perché derivato da valore tainted (42 untainted)

```
void foo()
{
    int x;

    x = read_password();
    use(x);
    x = 0; // clear password
    untrusted();
    return;
}
```

Poiché il valore 0 memorizzato in x  
non viene mai utilizzato,  
L'assegnamento (x = 0) viene  
rimosso (silenziosamente)  
dall'ottimizzazione statica dead-  
code elimination

### Back end

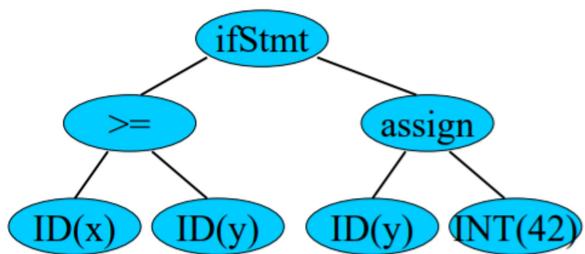
Traduce il codice intermedio nel linguaggio della macchina ospite o della macchina virtuale

Necessaria una conoscenza profonda della macchina ospite o virtuale

Usa le risorse della macchina in modo effettivo

- Input

```
if (x >= y)
y = 42;
```



- Output

```
mov eax,[ebp+16]
cmp eax,[ebp-8]
jl L17
mov [ebp-8],42
L17:
```

# 8 - Gestione dei dati

martedì 4 gennaio 2022 11:14

## DATI

*Livello di progetto: organizzazione dell'informazione*

- Dati e tipi diversi per concetti diversi
- Specifica e verifica di comportamenti
- Meccanismi esplicativi dei linguaggi per l'astrazione sui dati

*Livello di programma: identificano e prevengono errori*

- Controllo dei tipi

*Livello di implementazione: permettono alcune ottimizzazioni*

- Informazioni necessarie alla macchina per allocare spazio di memoria

## Classificazione dei dati:

- **Denotabili:** possono essere associati a un nome
- **Esprimibili:** possono essere il risultato della valutazione di una espressione
- **Memorizzabili:** se possono essere memorizzati (e modificati) in una variabile

## Tipi di dato - domini semanticici

- **Tipi di dato di sistema**  
definiscono lo stato e le strutture dati utilizzate nella simulazione/interpretazione del comportamento delle primitive del linguaggio (es run time stack)
- **Tipi di dato di programma**  
tipi primitivi del linguaggio e tipi che l'utente può definire

Un tipo di dato è una collezione di valori rappresentati da opportune strutture dati e un insieme di operazioni per manipolarli

Importanti per noi semantica e implementazione

## Implementazione: descrittori di dato

Rappresentare una collezione di valori utilizzando quanto ci viene fornito da un linguaggio della macchina ospite

Problema: riconoscere il valore e interpretare correttamente la rappresentazione

- È necessario associare alla rappresentazione concreta un'altra struttura che contiene la descrizione del tipo (descrittore di dato) che viene utilizzato ogni qualvolta si applica al dato un'operazione
  - Per type-checking dinamico
  - Per applicare l'operatore giusto

```
type exp =
  (* AST*)
  | Eint of int
  | Ebool of bool
```

```
type evT =
  (*Valori run-time*)
  | Int of int
  | Bool of bool
```

*Se l'informazione sui tipi è conosciuta completamente a tempo di compilazione*

- Si possono eliminare i descrittori di dato
- Il type checking è effettuato totalmente dal compilatore (statico)

*Se l'informazione sui tipi è a tempo di esecuzione*

- Sono necessari i descrittori per tutti i tipi di dato
- Type checking dinamico

*Se l'informazione sui tipi è conosciuta solo parzialmente a tempo di compilazione*

- I descrittori di dato contengono solo l'informazione dinamica
- Type checking in parte statico in parte dinamico

## Definizione e implementazione dei tipi di dato

**SCALAR:** bool, char, interi, reali

- Booleani
  - val: true, false
  - op: or, and, not, condizionali
  - rep: un byte
  - note: C non ha un tipo bool
- Caratteri
  - val: a,A,b,B, ..., è,é,ë, ; , ' ...
  - op: uguaglianza; code/decode; dipendono dal linguaggio
  - Rep: un byte (ASCII) o due byte (UNICODE)
- Interi
  - val: 0,1,-1,2,-2,...,maxint
  - op: +, -, \*, /, %, ...
  - repr: alcuni byte (2 o 4); complemento a due
  - note: interi e interi lunghi (anche 8 byte);
  - **limitati problemi nella portabilità quando la lunghezza non è specificata nella definizione del linguaggio**
- Reali
  - val: valori razionali in un certo intervallo
  - op: +, -, \*, /, ...
  - repr: alcuni byte (4); virgola mobile
  - note: reali e reali lunghi (8 byte);
  - **problematici per la portabilità quando la lunghezza non è specificata nella definizione del linguaggio**

## VOID (unit)

Il tipo void ha un solo valore (), non ha operazioni e serve per definire il tipo di operazioni che modificano lo stato senza restituire valori

## TIPI COMPOSTI

**Record:** collezione di campi, ciascuno di un diverso tipo. **Memorizzabili, esprimibili e denotabili**

**Record varianti.** Record in cui solo alcuni campi (mutualmente esclusivi) sono attivi in un dato istante

**Array:** funzione da un tipo indice ad un altro tipo, stringhe -> array di caratteri

**Insieme:** sottoinsieme di un tipo base

**Puntatore:** riferimento ad un oggetto di un altro tipo

## IMPLEMENTAZIONI

**RECORD:** memorizzazione sequenziale dei campi.

- Allineamento dei campi alla parola (32/64 bit) (**accesso semplice, spreco di memoria**)
- Padding o packed record (**disallineamento, accesso più costoso**)

```

struct MixedData
{
    char Data1; /* 1 byte */
    char Padding1[1]; /* 1 byte for the following 'short'
                        to be aligned on a 2 byte boundary*/
    short Data2; /* 2 bytes */
    int Data3; /* 4 bytes - largest structure member */
    char Data4; /* 1 byte */
    char Padding2[3]; /* 3 bytes to make total size of
                        the structure 12 bytes */
};

Si mettono dei byte di padding per far tornare decentemente la grandezza del record

```

```

struct MixedData /* field-reordering */
{
    char Data1;
    char Data4;
    short Data2;
    int Data3;
};


```

Allineamento alla parola:

```

// effettivo "memory layout" (C COMPILER)
struct x_{
    char a;           // 1 byte
    char _pad0[3];   // padding 'b' su 4 byte
    int b;           // 4 byte
    short c;         // 2 byte
    char d;           // 1 byte
    char _pad1[1];   // padding sizeof(x_)
                      // multiplo di 4
}

```

## ARRAY

Collezioni di dati omogenei, funzione da tipo indice a tipo elementi, indice discreto, elemento qualsiasi tipo

Possono essere multidimensionali

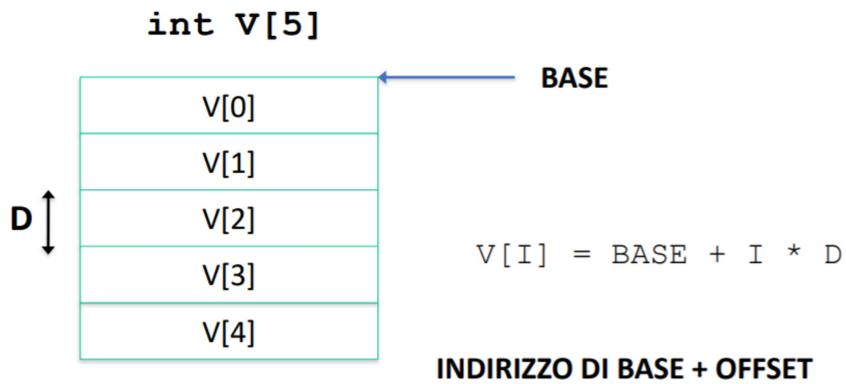
Operazioni: selezione elemento !! La modifica non è un'operazione sull'array, ma sulla locazione modificabile che memorizza un elemento di un array !!

## IMPLEMENTAZIONE

Elementi memorizzati in locazioni contigue

Formula di accesso:

Array V[N] of elem\_type; V[I] = base + c\*I (dove c è la dimensione per memorizzare un elem\_type)



### D dimensione in byte del tipo di base

Javascript non ha controlli per un eventuale array out of bounds, ma allora semplicemente altra memoria

C non ha controlli a runtime ma accede alla casella di memoria out of bounds (eventualmente segmentation fault) -> BufferOverflow

Java ha controllo a runtime -> solleva eccezione `ArrayIndexOutOfBoundsException`

Supponiamo di avere dichiarato (nel linguaggio C) un array di dimensione 4

`char buffer[4];`

Quale è il risultato dell'esecuzione dell'istruzione seguente?

`buffer[4] = 'a';`

Può succedere di tutto!!!!

**In un'ottica di cybersecurity, se un attaccante è in grado di controllare il valore 'a', il comportamento effettivo del programma dipende dalla scelta dell'attaccante**

Se siamo **fortunati** l'esecuzione potrebbe generare un **SEGMENTATION FAULT**:

Se siamo sfortunati si potrebbe attivare una esecuzione di un codice remoto malevolo **remote code execution (RCE)** ...  
un effetto non particolarmente gradevole

La mancanza di type safety è la fonte principale degli errori di programmazione e può permettere a eventuali attaccanti di sfruttare bug per eseguire codice malevolo o prendere il controllo del flusso di controllo

### FUNZIONI COME DATI

Higher order functions

Sulle funzioni possono essere eseguite svariate operazioni possibili anche per altri dati non funzionali, i linguaggi di programmazione prevedono quindi meccanismi linguistici e sistemi di tipo opportuni

### Inferenza di tipo su una funzione

Considerare una collezione di tipi base non interpretati, questi tipi non hanno operazioni, sono dei placeholder per tipi specifici. I tipi base non interpretati sono delle variabili di tipo che possono essere sostituite o istanziate con altri tipi

Strategia di risoluzione:

1. Progettare un algoritmo syntax-based, che assegna ad ogni espressione un tipo (*t*) che può contenere variabili di tipo, generando contemporaneamente un insieme di vincoli (C) (sistema di equazioni)

2. Risolvere il sistema di equazioni C

$$e: \text{fun}(x) = x + 1$$

$$\tau: X \rightarrow X \quad C: (X = \text{Int})$$

Può essere estesa la definizione di beta riduzione per gestire sostituzioni tra variabili di tipo e tipo

### Definizione

- Un insieme di vincoli C è un insieme di equazioni  $\{\rho_i = \kappa_i | i = 1, \dots n\}$  dove  $\rho_i$  e  $\kappa_i$  sono tipi contenenti variabili di tipo
- Un insieme di vincoli C è soddisfatto se esiste una sostituzione  $\sigma$  rende identiche tutte le equazioni  $\rho_i = \kappa_i$  ( $\sigma(\rho_i) = \sigma(\kappa_i)$ )

Indichiamo con  $\sigma(\rho)$  l'applicazione della sostituzione  $\sigma$  al tipo  $\rho$

- ESEMPIO:

$$C = \{X \rightarrow \text{Int} = Y, \quad X = \text{Int}\}$$

$$\begin{aligned}\sigma(X) &= \text{Int} \\ \sigma(Y) &= \text{Int} \rightarrow \text{Int}\end{aligned}$$

Un programma ben tipato si comporterà correttamente indipendentemente dai tipi concreti sostituiti alle sue variabili di tipo

Durante la fase di analisi sintattica (parsing) il compilatore annota ogni lambda fun x = e con una variabile fresca di tipo fun x:X = x con *X variabile di tipo fresca*, il compilatore poi esegue l'inferenza di tipo per determinare i valori più generali per le variabili di tipo che rendono il programma ben tipato

Codice del type checker sulle slide

### Unificazione

Processo algoritmico di risoluzione di equazioni tra espressioni simboliche (che contengono variabili). Una soluzione di un problema di unificazione è una sostituzione che assegna un valore simbolico ad ogni variabile delle espressioni del problema.

L'algoritmo di unificazione determina, per un dato problema, un insieme di sostituzione completo e minimo. (determina tutte le sue soluzioni senza soluzioni ridondanti)

### POLIMORFISMO

Una funzione è polimorfa se ha la caratteristica di essere applicabile ad argomenti di tipo diverso (ad es. ocaml funzione identità)(es. funzione map di ocaml)

### Overloading

Stesso nome, parametri di tipo diverso, diverso tipo del risultato

```
int foo(int x);  
float foo(float x);  
double foo(double x, double y);
```

### Subsumption

Alcuni tipi hanno caratteristiche migliori di altri, nel senso che un valore di un tipo può sempre essere usato con sicurezza dove ci si aspetta un valore dell'altro tipo

Formalmente

1. Una relazione di sottotipo,  $S < T$ , S è un sottotipo di T
2. Regola di subsumption che afferma che, se  $S < T$  allora ogni valore di tipo S può anche essere considerato di tipo T

(il tipo di un record con  $n + k$  campi è un sottotipo del tipo del record di  $k$  campi, il tipo di un record con più campi esprime un vincolo più forte sui valori, pertanto descrive meno valori)

**Top:** costante che è supertipo di ogni tipo (tipo classe Object in Java)

Funzioni: subsumption

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

#### Intuizione

Consideriamo una funzione  $f$  di tipo  $S_1 \rightarrow S_2$ ,  $f$  prende in ingresso valori di tipo  $S_1$  e potrà anche operare con argomenti di un qualsiasi sottotipo  $T_1$  di  $S_1$ .

Il tipo di  $f$  ci dice anche che la funzione produce come risultato valori di tipo  $S_2$ ; questi risultati possono anche appartenere a qualunque supertipo  $T_2$  di  $S_2$ .

Pertanto una funzione di tipo  $S_1 \rightarrow S_2$  può anche essere vista come avente tipo  $T_1 \rightarrow T_2$ .

La relazione di sottotipo

$$S <: Top \quad S <: S \quad \frac{S_1 <: S_2, S_2 <: S_3}{S_1 <: S_3}$$

La relazione di sottotipo è riflessiva e transitiva (preordine) con elemento massimo

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$

La relazione di sottotipo è **contravariante** nella parte sinistra del tipo freccia e **covariante** nella parte destra.

**Controvariante:** inverte la relazione d'ordine

**Covariante:** rispetta l'ordine

#### Puntatori

Valori: riferimenti, null

Operazioni:

- Creazione, funzione di libreria che alloca e restituisce un puntatore (malloc...)
- Dereferenziazione: accesso al dato puntato \*p
- Test di uguaglianza: in genere si uguaglia con null

Dal punto di vista dei tipi:

Ref T è un puntatore a una cella che contiene un valore di tipo T

#### Aliasing

Vairabili diverse che riferiscono la stessa cella di memoria

Rende più difficoltosa l'analisi del comportamento di un programma

Ottimizzazioni del codice dipendono dall'ipotesi che il compilatore sappia a quali entità potrebbe fare riferimento una determinata operazione

Alcuni linguaggi non permettono l'aliasing, ma è comunque una funzione utile (operazioni efficienti su array, risorse condivise in sistemi concorrenti)

*Problematiche che rendono difficile la comprensione e l'analisi di programmi complessi:*

*Global Variables Considered Harmful*  
(Wulf, Show 1973)

*Effetti laterali* (accessi nascosti di comportamento)

Aliases

Modificabilità incrementale del codice

Separare l'uso di una struttura di dati in un programma dalla forma particolare della struttura di dati stessa (evitare dipendenza tra l'uso di una struttura e la sua implementazione)  
Safety from bugs, ease of understanding, readiness for change.

### **Principi di astrazione dei dati**

- **Astrazione:** nascondere i dettagli di basso livello
- **Modularità:** dividere un sistema in componenti che possono essere gestiti separatamente
- **Incapsulamento:** barriera di astrazione intorno a un modulo per renderlo responsabile solo del proprio comportamento interno, senza essere influenzato da altri moduli
- **Information hiding:** nascondere i dettagli dell'implementazione di un modulo dal resto del sistema in modo che possano essere cambiati senza cambiare altre parti del sistema
- **Separation of concerns:** ogni modulo è responsabile di una caratteristica (concern), non viene distribuita su più moduli

### **Tipi di dato astratti (Abstract Data Types)**

Insieme di dati e una collezione di operazioni per operare sui dati di quel tipo

- Estendibili: meccanismi linguistici per costruire nuovi tipi di dato astratto
- Astratti: rappresentazione nascosta agli utenti del dato
- Incapsulati: si opera solo con operazioni fornite dall' ADT
- Distinzione tra specifica e implementazione

### **Specifiche:**

La specifica descrive il tipo di dati e le operazioni senza fornire dettagli di implementazione -> semantica delle operazioni

Fornisce le informazioni per usare il tipo nello sviluppo di programmi

Descrivono il cosa delle funzionalità di un modulo di un programma lasciando il come a chi deve realizzare le funzionalità offerte dal modulo

```

Stack <E>
Signature //Operazioni significative
  new      : -> STACK
  push     : E, STACK -> STACK
  top      : STACK -> E
  pop      : STACK -> STACK
  isEmpty  : STACK -> Bool
  undef_e  : -> E      //Completezza
  undef_s  : -> STACK

Axioms //Proprietà astratte da rispettare
\forall e: E, stk: STACK
  top(push(e, stk)) = e ;
  top(new) = undef_e ;
  pop(push(e, stk)) = stk ;
  pop(new) = undef_s ;
  isEmpty(new) ;
  ~isEmpty(push(e, stk))

```

Gli ADT ci forniscono un modo per definire formalmente i moduli riutilizzabili in un modo matematicamente valido, preciso e non ambiguo

Devono avere una struttura FAMED: Formale, ampiamente applicabile, minimo, estensibile e dichiarativo

Tipici ADT: coda, hash map, linked list, stack, alberi, liste

Le nozioni alla base degli ADT costituiscono gli aspetti di fondo della OOP

# 9 - Object Oriented Programming

mercoledì 5 gennaio 2022 11:15

## Modularità dei programmi

Modi per modularizzare i programmi:

- Astrazione procedurale (scomporre il programma in funzioni)
- ADT
- Compilazione separate e linking

## Livelli di astrazione

La possibilità di modularizzare i programmi consente di progettare e sviluppare un programma per livelli di astrazione

Tutti i sistemi informatici complessi sono organizzati in livelli di astrazione (es. linguaggi: source code -> bytecode -> assembler)

Sviluppare un programma complesso in modo modulare consente di suddividere il lavoro tra sviluppatori diversi, si definiscono le specifiche e ognuno sviluppa la propria parte

## Il paradigma a oggetti

Sistema software = insieme di oggetti cooperanti.

Gli oggetti sono caratterizzati da uno **stato** e un insieme di **funzionalità**; gli oggetti cooperano scambiandosi messaggi

**Stato di un oggetto:** rappresentato da un gruppo di attributi, idealmente lo stato di un oggetto non è accessibile agli altri oggetti (information hiding)

**Stato del programma:** insieme degli stati degli oggetti che lo compongono + strutture dati di sistema necessarie per l'esecuzione.

**Funzionalità di un oggetto:** rappresentate da un gruppo di metodi/funzioni che l'oggetto mette a disposizione agli altri oggetti; definiscono il **comportamento** dell'oggetto:

- Invio di un messaggio -> chiamata di metodo
- Risposta ad un messaggio -> restituzione del risultato

Altre caratteristiche degli oggetti:

- Identità
- Ciclo di vita
- locazione

## OOP concetti:

- ▶ **Incapsulamento** (già detto) e **Astrazione** (ragionare sul comportamento di un oggetto senza conoscerne la rappresentazione interna)
- ▶ **Interfaccia** (che cosa un oggetto mette a disposizione degli altri)
- ▶ **Ereditarietà** (come un oggetto può fare proprie le funzionalità di un altro oggetto, ad esempio **estendendolo**)
- ▶ **Principio di sostituzione** (quando un oggetto può essere **usato al posto di un altro** in maniera trasparente e controllata)
- ▶ **Polimorfismo** (come un oggetto può **processare** altri oggetti indipendentemente anche di «tipi» diversi)

(presenti in tutti i linguaggi obj oriented)

## Costrutti linguistici:

- Object based
- Class based

## *Object based:*

Gli oggetti vengono trattati nel linguaggio in maniera simile ai record  
I campi possono essere associati a funzioni  
Una funzione in un oggetto (metodo) può accedere ai campi dell'oggetto stesso tramite this  
Per creare un oggetto c'è bisogno di un **costruttore**

Consente al programmatore di lavorare con gli oggetti in modo flessibile.  
Rende difficile predire quale sarà il tipo di un oggetto

#### ***Class based:***

Prevede un concetto di classe.

Una classe definisce il contenuto degli oggetti di un certo tipo

Gli oggetti vengono creati successivamente come istanze di una certa classe

Richiede di implementare classi prima di oggetti

Consente di fare controlli statici sugli oggetti -> il tipo dell'oggetto è legato alla classe di cui è istanza (nominal typing)

#### **Ereditarietà**

Funzionalità che consente di definire una classe sulla base di un'altra esistente

I linguaggi object based per ogni oggetto mantengono una lista di prototipi (oggetti da cui esso eredita funzionalità)

I linguaggi class based consentono di definire una classe come estensione di un'altra. La nuova classe eredita tutti i membri della precedente e può aggiungerne altri o ridefinirne.

#### **Subtyping**

Un oggetto B che è estensione di un altro oggetto A dovrebbe poter essere usato dovunque si possa usare A

##### ***Object based:***

**Subtyping strutturale:** un oggetto B è sottotipo di un oggetto A se contiene almeno i membri pubblici di A.

È più flessibile, non è necessario specificare chi estende chi

Favorisce il polimorfismo

##### ***Class based:***

**Subtyping nominale:** un tipo di un oggetto corrisponde alla classe da cui è stato istanziato (nome della classe = tipo).

Un tipo-classe B è sottotipo di un tipo-classe A se la classe B è stata definita come estensione della classe A

Solo classi che dichiarano specificatamente di estendere altre.

Più semplice verificare per l'interprete.

#### **Oggetti in Ocaml**

Un oggetto in ocaml è un valore costituito da campi e metodi

Object based

Il tipo è dato dai metodi che contiene

```
(* oggetto che realizza uno stack *)
let s = object

  (* campo mutabile che contiene la rappresentazione dello stack*)
  val mutable v = [0; 2] (* Assumiamo per ora inizializzato non vuoto *)

  (* metodo pop *)
  method pop =
    match v with
    | hd :: tl ->
      v <- tl;
      Some hd
    | [] -> None

  (* metodo push *)
  method push hd =
    v <- hd :: v
end ;;
```

### TIPO INFERITO (object-type, solo metodi):

```
val s : < pop : int option; push : int -> unit > = <obj>
```

L'invocazione di un metodo si fa con # notation s#pop

### Type weakening

Se inizializzo v come lista vuota:

### TIPO INFERITO

```
val s : < pop : '_weak option; push : '_weak -> unit > = <obj>
```

Il type checker non ha un tipo concreto noto al momento della dichiarazione quindi indebolisce temporaneamente il tipo inferito includendo delle variabili di tipo, al primo utilizzo viene istanziata definitivamente la variabile provvisoria a un tipo concreto

Gli oggetti possono essere costruiti tramite funzioni

```
(* funzione che costruisce oggetti inizializzati con init *)
let stack init = object
  val mutable v = init

  method pop =
    match v with
    | hd :: tl ->
      v <- tl;
      Some hd
    | [] -> None

  method push hd =
    v <- hd :: v
end ;;
```

### TIPO INFERITO

```
val stack : 'a list ->
  < pop : 'a option; push : 'a -> unit >
  = <fun>
```

La funzione stack è (veramente) polimorfa!

### Polimorfismo

Quando si definisce una funzione che prende un oggetto come parametro, il tipo dell'oggetto viene inferito dai metodi che vengono chiamati su di esso

```
let area sq = sq#width * sq#width ;;
val area : < width : int; .. > -> int = <fun>

let minimize sq = sq#resize 1 ;;
val minimize : < resize : int -> 'a; .. > -> 'a = <fun>

let limit sq = if (area sq) > 100 then minimize s
val limit : < resize : int -> unit; width : int; .. > -> unit = <fun>
```

Questa notazione indica che l'oggetto atteso come parametro deve contenere **almeno** il metodo width, ed eventualmente anche altro (espresso tramite puntini .. )

Structural subtyping:

Si applica la regola di **subsumption**

$$\frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T}$$

esattamente come nel caso dei record. Abbiamo:

```
< color : string;
  resize : int -> unit;    <:    < width : int >
  width : int >
```

Grazie a cui, nel nostro esempio, possiamo derivare

$$\Gamma \vdash quadrato : < width : int >$$

e concludere che l'oggetto `quadrato` può essere passato alla funzione `area`. Lo stesso vale per le funzioni `minimize` e `limit`.

La notazione

```
<width : int, .. >
```

Enfatizza lo structural subtyping

La funzione accetta un qualunque oggetto contenga almeno il metodo `width`

### Coercion di tipi oggetto

```
type shape = < area : float >
type square = < area : float; width : int >
```

E' chiaro che `square` sia un sottotipo di `shape` (contiene dei metodi in più), quindi è lecito pensare che ovunque si possa usare un oggetto di tipo `shape` si possa usare al suo posto un oggetto di tipo `square` (**PRINCIPIO DI SOSTITUZIONE**, ne ripareremo...)

Se provi però a usare `square` al posto di `shape` non funziona.

Serve una type coercion esplicita con l'operatore `:>`

```
let lis2 = ( square 5 :> shape ) :: lis1 ;;
val 12 : shape list = [<obj>; <obj>; <obj>]
```

`e :> t` forza il type checker a considerare `e` come tipo `t`, ovviamente `t` deve essere supertipo di `e` sennò non va bene!!!!!!!

Polimorfismo di oggetti e principio di sostituzione sono trattati in maniera diversa in ocaml perché il type checker non effettua conversioni implicite

In java è possibile fare type coercion da supertipo a sottotipo

Ocaml introduce anche costrutti di **classe**

► una classe si definisce con `class` e si istanzia con `new`

```
class istack = object (* classe per stack di interi *)
  val mutable v = [0; 2] (* inizializzato non vuoto *)
  method pop =
    match v with
    | hd :: tl ->
      v <- tl;
      Some hd
    | [] -> None
  method push hd =
    v <- hd :: v
end ;;

let s = new istack ;;
val s : istack = <obj>
s#pop ;;
- : int option = Some 0
```

Una classe può prevedere parametri di **costruzione** che vanno passati al momento dell'istanziazione

e di **tipo** che la rendono polimorfa

La definizione di una classe introduce anche un tipo con lo stesso nome, **alias** del tipo oggetto che si otterrebbe costruendo gli oggetti

```
let s = new stack ["pippo"] ;;
val s : string stack = <obj>
(* string stack è un alias per
   < pop : string option; push : string -> unit > *)
```

Ereditarietà

```
class sstack init = object (* classe per stack di stringhe *)
  inherit [string] stack init (* eredita da stack *)

  method concat =           (* aggiunge un nuovo metodo *)
    List.fold_left (^) v
end ;;

let b = new sstack [" ";"world!"] ;;
val b : sstack = <obj>
b#push "Hello" ;;
- : unit = ()
b#concat ;;
- : string = "Hello world!"
```

OCaml **combina** costrutti linguistici tipicamente **object-based** con costrutti tipicamente **class-based**

- ▶ I costrutti **object-based** favoriscono la definizione di **object-types** e lo **structural subtyping**
- ▶ La **non modificabilità della struttura degli oggetti** (a differenza di JavaScript) consente di effettuare **controlli di tipo** a tempo di compilazione
- ▶ I costrutti **class-based** favoriscono una naturale definizione di meccanismi di **ereditarietà** (e altro...)

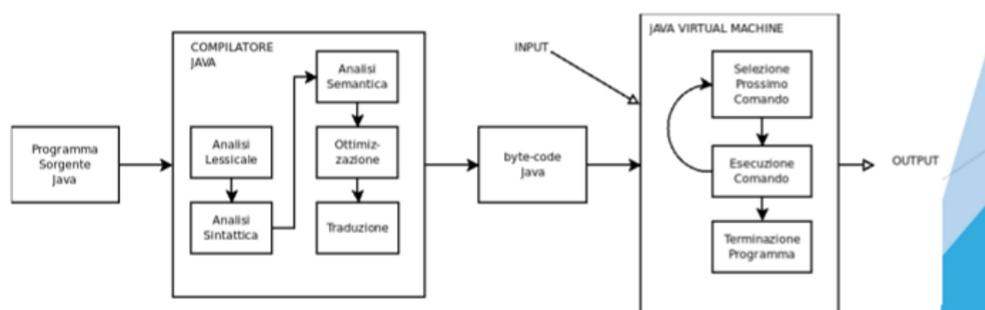
## JAVA

- Semplificare la programmazione
- Indipendenza della piattaforma
- Modularizzabilità
- Robustezza e sicurezza

Approccio **compilazione + interpretazione**, con un bytecode intermedio

Il bytecode è utilizzabile in ogni architettura per cui sia disponibile un interprete (JVM), senza ricompilare

Consente controlli statici (compilatore) + dinamici (interprete)



Tralascio le cose base di java che le ho fatte a reti

## **FUNZIONI ANONIME:**

- ▶ Si possono definire funzioni anonime (**lambda-espressioni**) con la seguente sintassi:

$x \rightarrow \text{espressione}$	$(x, y, \dots) \rightarrow \text{espressione}$
$x \rightarrow \{ \text{blocco} \}$	$(x, y, \dots) \rightarrow \{ \text{blocco} \}$

Metodo statico: può essere usato senza instanziare la classe

Se modifco l'interfaccia pubblica (nomi metodi e variabili pubbliche), devo modificare e ricompilare le classi che le usano

Public/private -> **incapsulamento**

**Interfaccia:** specifica astratta della classe: quali membri pubblici deve contenere la classe (sorta di ADT)

Tipo apparente: usato dal compilatore per fare i controlli (tipo statico)

Tipo effettivo: tipo a runtime dell'oggetto (tipo dinamico)

Il cast (coercion) tra classi e interfacce (o misto) si può fare se tra le due esiste una relazione di sottotipo

Upcast: implicito

Downcast: va esplicitato

Una classe può implementare più interfacce

Membri d'istanza: stato di ogni singolo oggetto e implementano operazioni che lavorano su tale stato

Membri statici: informazioni su operazioni di classe, variabili condivise tra tutti gli oggetti che sono istanze di quella classe e metodi che non operano sullo stato dei singoli oggetti

- ▶ Una **variabile d'istanza** è presente in memoria in tante copie quanti sono gli oggetti
- ▶ Una **variabile statica** è presente una sola volta in memoria, anche prima che sia creato il primo oggetto di quella classe
- ▶ Un **metodo d'istanza** è richiamato su un oggetto e può accedere sia alle variabili d'istanza (relative a quell'oggetto) che a quelle statiche
- ▶ Un **metodo statico** può accedere solo alle variabili statiche e può essere richiamato anche se non ci sono oggetti di quella classe

## **Cosa succede in memoria quando esegui un codice java**

- ▶ L'**AMBIENTE DELLE CLASSI** (o **WORKSPACE**), che contiene il codice dei metodi e le **variabili statiche** (di classe)
- ▶ Lo **STACK**, che contiene i **record di attivazione** dei metodi con le **variabili locali**
- ▶ Lo **HEAP**, che contiene gli **oggetti** (raggiungibili tramite **riferimenti**) con le loro **variabili d'istanza**

Esempio di esecuzione sulle slide oop\_02 p 53

## **EREDITARIETÀ E DYNAMIC DISPATCH**

Ereditarietà per estensione (extends), ereditarietà singola.

Si possono implementare più interfacce ma si può estendere una sola classe

Tutte le classi estendono implicitamente object

Metodo equals ridefinirlo se serve perché confronta i riferimenti

I metodi di string non modificano gli oggetti, ma ne restituiscono sempre dei nuovi

Le regole di visibilità valgono anche tra super-classe e sotto-classe, **protected** li rende visibili alle sotto classi ma non agli altri

### Classi astratte

Classi parzialmente definite.

Contiene almeno un metodo astratto (non implementato).

Una classe astratta non può essere istanziata, ma può essere estesa

**Scopo:** definire un tipo di oggetto comune a più classi, fornendo anche implementazione per metodi comuni

### Gerarchia di classi

Extends consente di creare una gerarchia di classi, rappresentabile come un albero

Ad ogni classe è associato un tipo di oggetto

**Nominal** subtyping => la gerarchia di classi è una rappresentazione della relazione di sottotipo <:

La regola subsumption (sostituzione del sottotipo) del sistema dei tipi automaticamente ci consente di ottenere un meccanismo di polimorfismo per sottotipo

$$\text{Proprietà transitiva } \frac{T_1 <: T_2 \quad T_2 <: T_3}{T_1 <: T_3}$$

$$\text{Subsumption } \frac{\Gamma \vdash_e \exp : S \quad S <: T}{\Gamma \vdash_e \exp : T}$$

$$\text{New } \Gamma \vdash_e \text{new } T() : T$$

$$\text{Assegnamento } \frac{\Gamma \vdash_e \exp : T}{\Gamma \vdash_c T x = \exp}$$

Con queste regole possiamo derivare:

$$\Gamma \vdash_c \text{Persona } p = \text{new StudTriennale}();$$

### Teorema

Sia <: la relazione di tipo inferita dalla gerarchia di classi di Java. Per ogni coppia di classi S e T tale che S <: T, ogni membro pubblico di T è anche membro pubblico di S

### Dimostrazione

Banale... segue dalla definizione di extends e implements e dalla transitività di <:

### Conseguenza

Structural subtyping <:s è una relazione più debole del Nominal Subtyping <:n

$$S <:n T \Rightarrow S <:s T \quad S <:s T \not\Rightarrow S <:n T$$

### Overloading, overriding e dynamic dispatch

Cosa succede quando si invoca un metodo?

Il metodo deve essere presente nella classe e i parametri attuali devono essere compatibili con i parametri formali del metodo

### Overloading di metodi

Si possono definire più metodi con lo stesso nome, ma con firme diverse

int minimo(int a, int b)

minimo(int, int)

double minimo(double x, double y)

minimo(double, double)

Se il metodo non viene trovato si ha un errore in fase di compilazione, ma potrebbe essere che il controllo è fatto sul tipo statico, quindi ci vuole un downcast esplicito, meglio se lo facciamo dopo

aver controllato il tipo dinamico con instanceof

```
Persona p = new Studente();  
if (p instanceof Studente) {  
    Studente s = (Studente) p;  
    int m = s.getMatricola();  
}
```

### Dynamic dispatch

A tempo di esecuzione la chiamata del metodo si può trovare in due casi:

- Il metodo è presente nella classe dell'oggetto
- Il metodo va cercato nelle superclassi

La JVM va a cercare il metodo dalla classe corrente risalendo la gerarchia (dynamic dispatch)

La ricerca del metodo parte dalla classe che corrisponde al **TIPO EFFETTIVO** dell'oggetto.

La JVM trova l'indicazione sul tipo effettivo nel **descrittore del dato** (oggetto) in memoria, inizializzato al momento della creazione dell'oggetto

stud: <class Studente>	
nome:	"Mario"
indirizzo:	"Via..."
matricola:	14421
anno:	2

### Overriding

Sottoclassi che ridefiniscono metodi delle superclassi, la JVM esegue il primo metodo che incontra risalendo la gerarchia

Visitare l'albero della gerarchia ad ogni chiamata di metodo è inefficiente

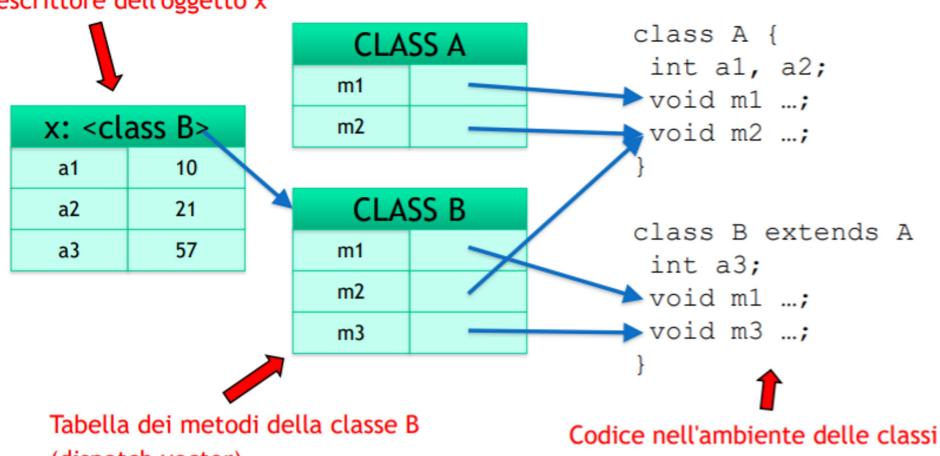
La JVM adotta una soluzione che si basa su

- Tabelle di metodi (dispatch vector) (tabella con puntatori ai metodi)
- Sharing strutturale

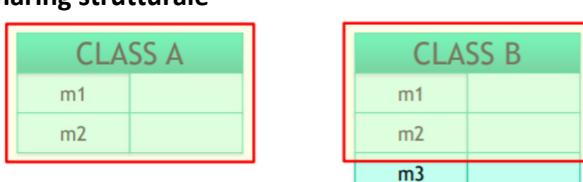
La tabella della sottoclasse riprende la struttura della tabella della superclasse aggiungendo righe per i nuovi metodi

## Nella JVM (a runtime)

Descrittore dell'oggetto x

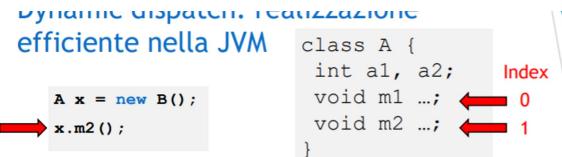


Sharing strutturale



Con questa soluzione, l'operazione di **dispatching dei metodi** la si può risolvere **STATICAMENTE**

- Il **compilatore determina l'offset** del metodo nella tabella (corrisponde alla posizione in tabella)
- L'offset è determinato sul **tipo apparente** dell'oggetto
- A tempo di esecuzione la JVM accede alla tabella della classe che è **tipo effettivo** usando quell'offset
- Anche se il tipo effettivo dovesse essere **diverso** da quello apparente, corrisponderà comunque ad una sua **sottoclasse**, e grazie allo **sharing strutturale** la JVM accederà comunque al **metodo giusto**
- Se la sottoclasse ha fatto **overriding** del metodo, il **puntatore** che si troverà in tabella **riferirà alla nuova versione** del metodo



Quindi niente visita dell'albero delle classi, ma accesso diretto tramite offset e puntatore

I metodi accedono alle variabili d'istanza tramite un riferimento implicito (this) all'oggetto stesso che viene esplicitato a tempo di esecuzione.

I metodi statici non possono accedere a variabili d'istanza, a run time sono implementati con dispatch vector dedicato a cui si accede tramite l'operazione invokestatic nel bytecode

### PRINCIPIO DI SOSTITUZIONE DI LISKOV

Fasi della progettazione

1. **Definizione** della **gerarchia di classi** e interfacce da realizzare
2. **Identificazione** dei **membri pubblici** di ogni classe
3. **Definizione** delle **SPECIFICHE** di ogni metodo pubblico (condizioni sui parametri, sul risultato e comportamento atteso del metodo)
4. **Implementazione** di **programmi di testing** delle singole classi sulla base di quanto definito (membri pubblici e specifiche)
5. **Definizione** dei **membri privati** seguendo il **principio di encapsulamento**
6. **Implementazione** del **codice delle classi**, da verificare con i test già sviluppati

### Definizione delle specifiche

- Precondizioni (REQUIRES): condizioni sui parametri e sul valore delle variabili che devono essere soddisfatte all'inizio dell'esecuzione del metodo
- Postcondizioni (EFFECTS): condizioni sul risultato e valori delle variabili che devono essere soddisfatte al termine dell'esecuzione assumendo che le precondizioni valgano

```
public boolean add(int elem) throws FullSetException {  
    // REQUIRES: numero di elementi contenuti nell'insieme minore di capienza  
    // EFFECTS: se elem non è presente nell'insieme lo aggiunge e restituisce true,  
    //           restituisce false altrimenti  
}
```

Tester epr collaudare se i metodi rispettano la relazione tra precondizioni e postcondizioni

### Principio di sostituzione di liskov

Un oggetto di un sottotipo può sostituire un oggetto del supertipo senza influire sul comportamento dei programmi che usano il supertipo

Esprime una relazione tra i comportamenti degli oggetti dei due tipi

La relazione deve valere per tutti i possibili contesti in cui la sostituzione può avvenire

I contesti possibili sono infiniti e sono programmi -> verificare se vale il LSP tra due classi è un problema indecidibile

Il principio di sostituzione di Liskov nella pratica si traduce in un **insieme di regole** da seguire:

► **la regola della segnatura**

- gli oggetti del sotto-tipo devono avere tutti i metodi del super-tipo
- le segnature (firme) dei metodi del sotto-tipo devono essere compatibili con le segnature dei corrispondenti metodi del super-tipo

► **la regola dei metodi**

- le chiamate dei metodi del sotto-tipo devono comportarsi come le chiamate dei corrispondenti metodi del super-tipo

► **la regola delle proprietà**

- il sotto-tipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del super-tipo

### Regola della segnatura

Garantita dal compilatore

In caso di overriding il metodo deve avere la stessa firma; può avere un return type più specifico (sottotipo) di quello della superclasse

se il tipo del metodo della super-classe è  
 $S \rightarrow T_1$   
il tipo del metodo della sotto-classe può essere:  
 $S \rightarrow T_2$   
con  $T_2 <: T_1$  (**covariante**)

### Regola dei metodi

Un sotto tipo può indebolire la precondizione e rafforzare la postcondizione dei metodi riscritti

- **regola della pre-condizione**  $pre_{super} \Rightarrow pre_{sub}$
- **regola della post-condizione**  $(pre_{super} \&& post_{sub}) \Rightarrow post_{super}$

## JAVA GENERICS

Meccanismo di astrazione linguistica che consente la definizione di classi e metodi parametrici rispetto al tipo che utilizzano, algoritmi che si applicano in contesti diversi ma in cui il funzionamento generale si applica indipendentemente dal contesto

### Varianza per tipi

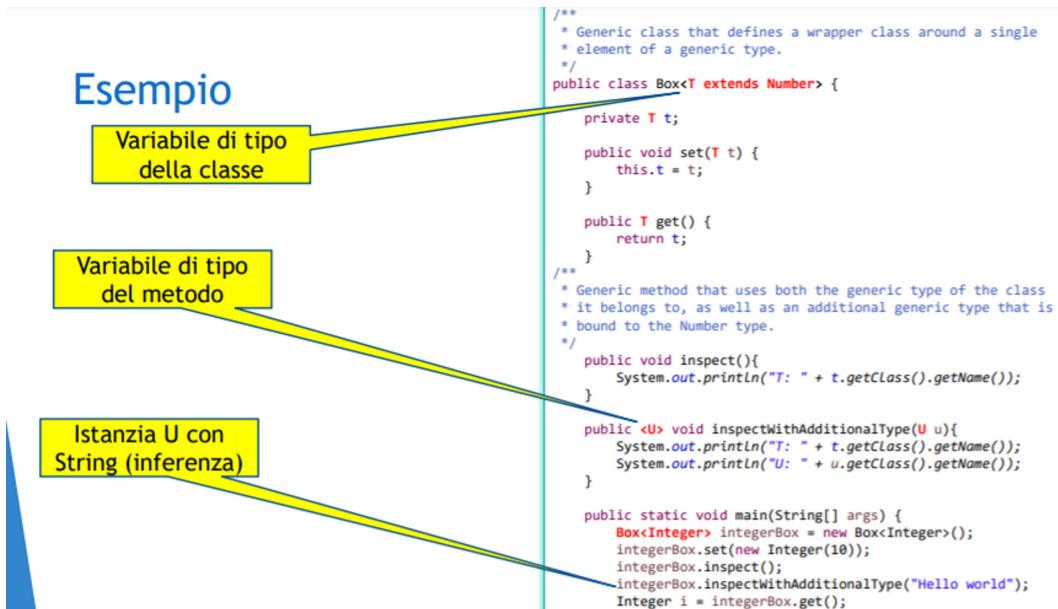
Supponiamo che  $A(T)$  sia un opportuno tipo definito usando il Tipo  $T$ :

- $A$  è **covariante** se  $T <: S$  implica  $A(T) <: A(S)$ ,
- $A$  è **contravariante** se  $T <: S$  implica  $A(S) <: A(T)$ ,
- $A$  è **bivariante** se è sia covariante che contravariante,
- $A$  è **invariante** se non è covariante e contravariante

### Metodi generici

I metodi di una classe generica possono usare le variabili di tipo dichiarate dalla classe e possono dichiarare dei propri tipi generici

## Esempio



Wildcard ?: tipo sconosciuto

Si usa quando si usa un tipo esattamente una volta ma non si conosce il nome

### Producer Extends, Consumer Super

Si usa ? Extends T nei casi in cui si vogliono ottenere dei valori

Si usa ? Super T nei casi in cui si vogliono inserire valori+

Non si usa quando bisogna ottenere e produrre valori

### ? vs Object

? Tipo particolare anonimo

```
void printAll(List<?> lst) {...}
```

Quale è la differenza tra `List<?>` e `List<Object>`?

- possiamo istanziare ? con un tipo qualunque: `Object`, `String`, ...
- `List<Object>` è più restrittivo: non posso passare un `List<String>`

Quale è la differenza tra `List<Foo>` e `List<? extends Foo>`

- nel secondo caso il tipo anonimo è un sottotipo sconosciuto di `Foo`

### Java array

Sappiamo bene come operare con gli array in Java ... Vero?

Analizziamo questa classe:

```
class Array<T> {  
    public T get(int i) { ... "op" ... }  
    public T set(T newVal, int i) { ... "op" ... }  
}
```

Domande: Se `Type1` è un sottotipo di `Type2`,

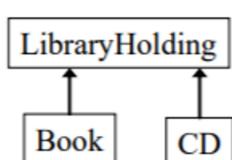
- ▶ quale è la relazione tra `Array<Type1>` e `Array<Type2>??`
- ▶ quale è la relazione tra `Type1[]` e `Type2[]??`

Come si comprano i tipi generici con gli array

Gli array sono degli oggetti e possiamo definirli di diversi tipi, nel caso del tipo array vale la covariante, un array di integer è un sottotipo di un array di number

È in contrasto su quello detto prima (nelle slide).

Usando gli array posso ritrovarmi nella situazione di errore ("del cane e del gatto")



posso mettere un CD in books?

- ▶ Il tipo dinamico (effettivo) è un sottotipo di quello statico (apparente)
  - violato nel caso di `Book b`
- ▶ La scelta di Java
  - ogni array "conosce" il suo tipo dinamico (`Book[]`)
  - modificare a (run-time) con un supertipo determina **ArrayStoreException**
- ▶ pertanto `replace17` solleva una eccezione
  - *Every Java array-update includes run-time check*
    - ✓ (dalla specifica della JVM)
  - **Morale: fate attenzione agli array in Java**

Tutti i tipi generici sono trasformati in Object nel processo di compilazione (**type erasure**)

```
class Vector<T> {  
    T[] v; int sz;  
    Vector() {  
        v = new T[15];  
        sz = 0;  
    }  
<U implements Comparer<T>> void sort(U c) {  
    ...  
    c.compare(v[i], v[j]);  
    ...  
}  
...  
Vector<Button> v;  
v.addElement(new Button());  
Button b = v.elementAt(0);
```

→

```
class Vector {  
    Object[] v; int sz;  
    Vector() {  
        v = new Object[15];  
        sz = 0;  
    }  
    void sort(Comparer c) {  
        ...  
        c.compare(v[i], v[j]);  
        ...  
    }  
    ...  
    Vector v;  
    v.addElement(new Button());  
    Button b = (Button)v.elementAt(0);
```

- ▶ Il compilatore verifica l'utilizzo corretto dei generici
- ▶ I parametri di tipo sono eliminati nel processo di compilazione e il "class file" risultante dalla compilazione è un normale class file senza poliformismo parametrico

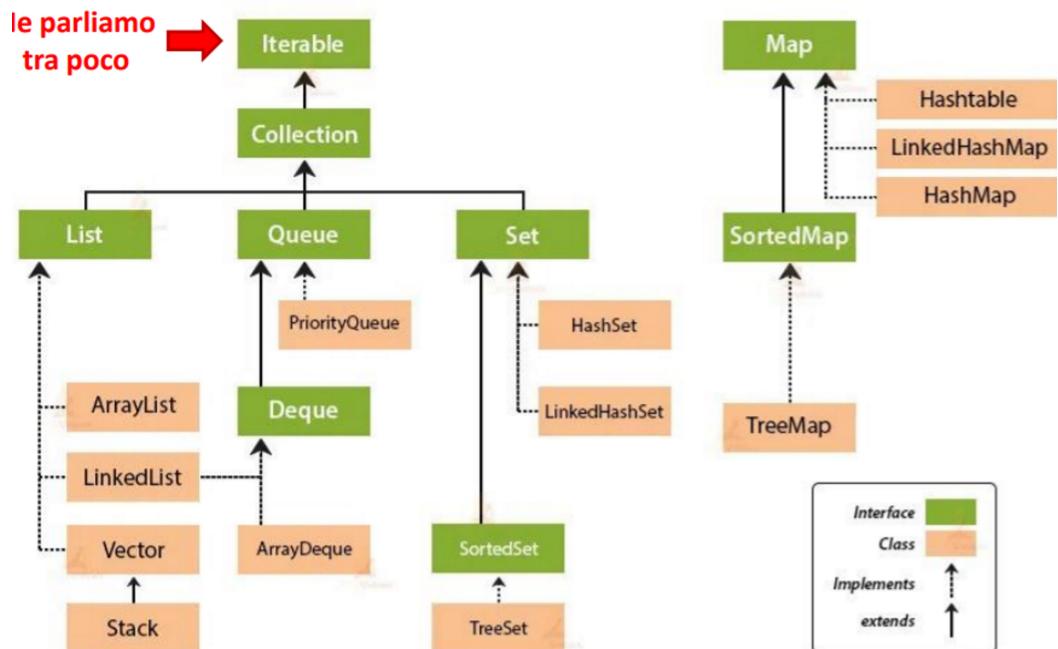
## JAVA COLLECTIONS FRAMEWORK

Manipolare gruppi di valori o oggetti dello stesso tipo

**Collezione:** gruppo di oggetti omogenei

**JCF:** definisce una gerarchia di interfacce e classi che realizzano una varietà di collezioni

## Collection Framework Hierarchy in Java



## Implementazione iteratori: IntSet

```
public class IntSet implements Iterable<Integer> {  
  
    private int[] a;  
    private int size;  
  
    public IntSet(int capacity) { ... }  
    public boolean add(int elem) throws FullSetException { ... }  
    public boolean contains(int elem) { ... }  
  
    public Iterator<Integer> iterator( ) {  
        return new IntSetIterator();  
    }  
  
    // INNER CLASS (CLASSE INTERNA)  
    private class IntSetIterator implements Iterator<Integer> {  
        ... (SEGUE) ...  
    }  
}
```

```

public class IntSet implements Iterable<Integer> {

    ...

    // INNER CLASS (CLASSE INTERNA)
    private class IntSetIterator implements Iterator<Integer> {
        private int curr=0;

        public boolean hasNext() { return (curr<size); }

        public Integer next() {
            if (curr>=size) throw new NoSuchElementException();
            return a[curr++];
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}

```

Ieratore è una classe che restituisce elementi di una collezione, un iteratore che ritorna elementi non di una collezione è un generatore (es random)

```

public class Primes implements Iterable<Integer> {
    public Iterator<Integer> iterator() {
        // EFFECTS: ritorna un generatore che produrrà tutti
        // i numeri primi (come Integers), ciascuno una
        // sola volta, in ordine crescente
    }
}

```

Come uso gli iteratori:

```

public class Primes implements Iterable<Integer> {
    public Iterator<Integer> iterator() {
        // EFFECTS: ritorna un iteratore che produrrà tutti i numeri
        // primi (come Integers) ciascuno una sola volta, in ordine
        // crescente
    }

    public static void printPrimes (int m) {
        // EFFECTS: stampa tutti i numeri primi minori o uguali a m
        // su System.out
        for (Integer p : new Primes()){
            if (p > m) return; // forza la terminazione
            System.out.println("The next prime is: " + p);
        }
    }
}

```

## Riassumendo: uso di iteratore



- Con successive chiamate di **next()** si visitano tutti gli elementi della collezione esattamente una volta
- **next()** lancia una **NoSuchElementException** esattamente quando **hasNext()** restituisce **false**
- L'ordine nel quale vengono restituiti gli elementi dipende dall'implementazione dell'iteratore
  - una collezione può avere più iteratori, che usano ordini diversi
  - per le collezioni lineari (come **List**) l'iteratore default rispetta l'ordine
- Si possono attivare più iteratori simultaneamente su una collezione
- Se invoco la **remove()** senza aver chiamato prima **next()** si lancia una **IllegalStateException()**
- Se la collezione viene modificata durante l'iterazione di solito viene invocata una **ConcurrentModificationException**

Creo un oggetto della classe **Primes** e poi faccio un **for each** su quell'oggetto

Questo **foreach** usa l'iteratore e tutte le volte che mi da un **p** va a generare un numero primo

**Primes** non è una collezione

```
public class Primes implements Iterable<Integer> {  
    public Iterator<Integer> iterator() { return new PrimeGen(); }  
    // EFFECTS: ritorna un generatore che produrrà tutti i numeri primi  
    // (come Integers) ciascuno una sola volta, in ordine crescente  
    private static class PrimeGen implements Iterator<Integer> {  
        // class interna statica  
        private List<Integer> ps; // primi già dati  
        private int p; // prossimo candidato alla generazione  
        PrimeGen() { p = 2; ps = new ArrayList<Integer>(); } // costruttore  
        public boolean hasNext() { return true; }  
        public Integer next() {  
            if (p == 2) { p = 3; ps.add(2); return new Integer(2); }  
            for (int n = p; true; n = n + 2)  
                for (int i = 0; i < ps.size(); i++) {  
                    int el = ps.get(i);  
                    if (n%el == 0) break; // non è primo  
                    if (el*el > n) { ps.add(n); p = n + 2; return n; }  
                }  
        }  
        public void remove() { throw new UnsupportedOperationException(); }  
    }  
}
```

Se il numero è divisibile per quelli presenti nell'array allora non è primo

Sennò è primo

Clean

```

public void printSet (IntSet set) {
    for (Integer i : set) System.out.println(i);
}
enhanced for (for-each) su IntSet

```

----- ASTRAE DA -----

hasNext(), next(), ... definizione di un iteratore

----- ASTRAE DA -----

int[] a; int size; rappresentazione dell'insieme

----- ASTRAE DA -----

heap, garbage collection, ... gestione della memoria nella JVM

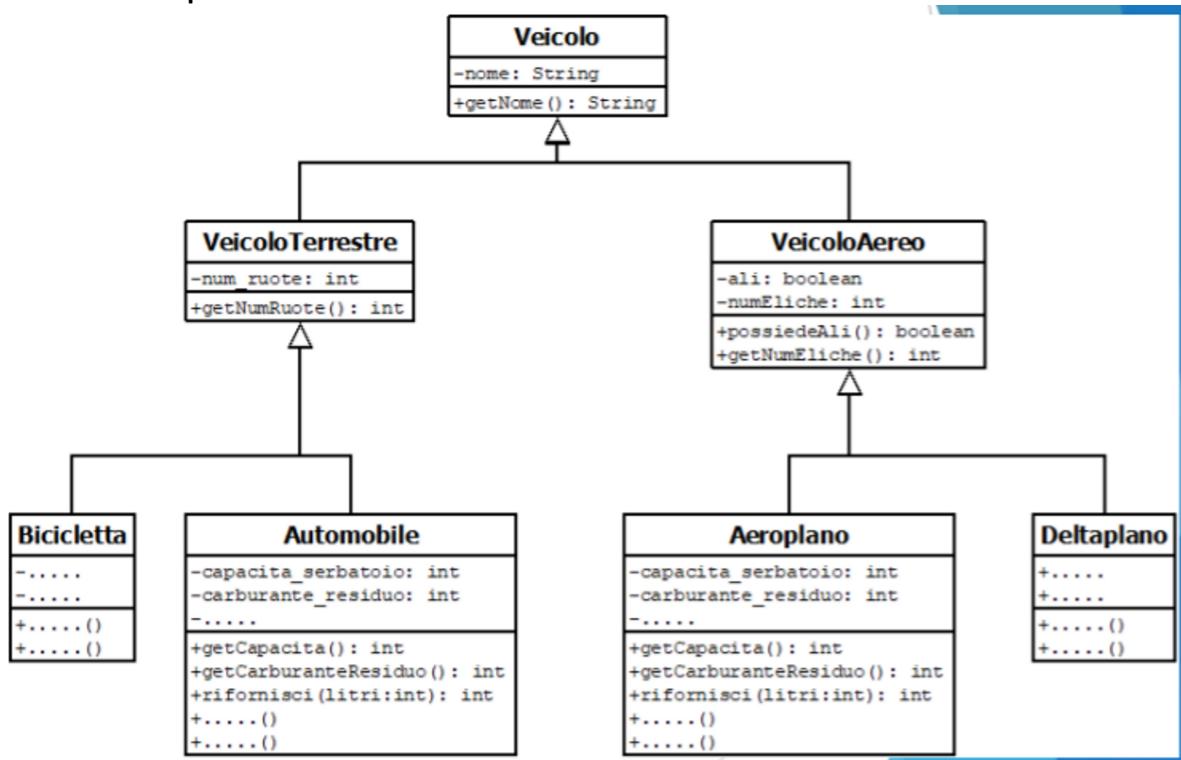
----- ASTRAE DA -----

malloc/new, free/delete, ... implementazione (in C++) della JVM

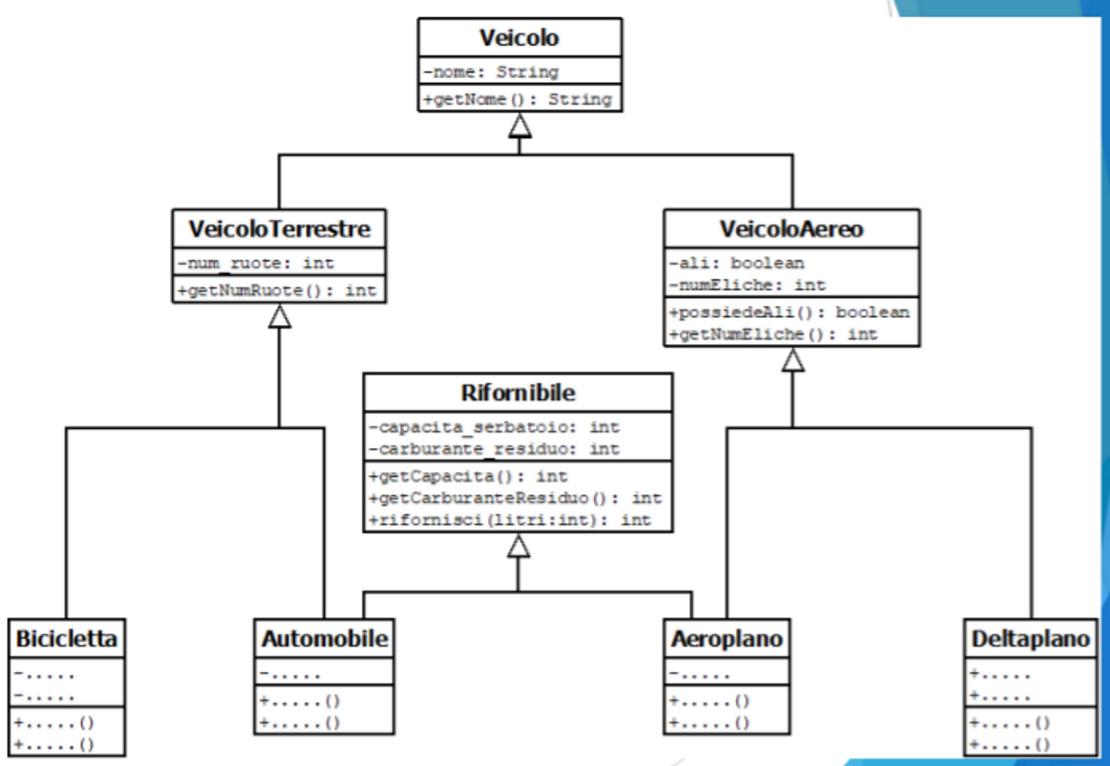
Scritto su programmazione funzionale

All'orale chiede i concetti di java che sono stati spiegati

### Ereditarietà multipla



In java non ho modo di mettere questo codice relativo a questi metodi in un unico posto, devo implementare due volte, ho del codice duplicato, posso metterci un'interfaccia ma ho comunque duplicazione del codice, mi servirebbe una superclasse comune a automobile e aereoplano -> rifornibile

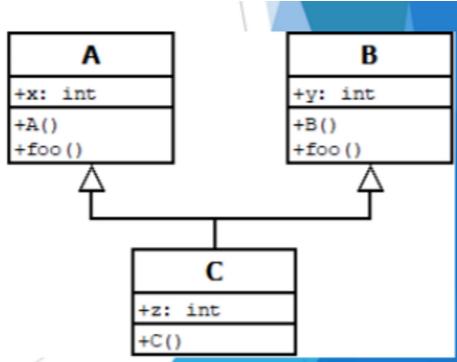


Non posso definirla come interfaccia sennò comunque devo ri-implementare il codice sia di qui che di qua

L'unica è l'ereditarietà multipla

*Perché l'ereditarietà multipla è un problema?*

Possibilità di ereditare due implementazioni diverse dello stesso metodo, quindi sono in conflitto

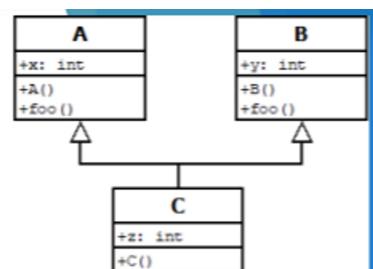


Dove è disponibile estendere più classi:

- C++ -> consente di disambiguare quale metodo implementare
- Java -> interfacce per definire supertipi senza definire superclassi
- Python -> gerarchia di classi come lista di classi (controllo statico)
- Dart -> composizione di classi (mixin) al posto di ereditarietà (mixin costrutto nuovo neanche troppo ben definito)

*(gli esempi sono tutti eseguibili su repl.it)*

### Esempio ereditarietà multipla in C++



```

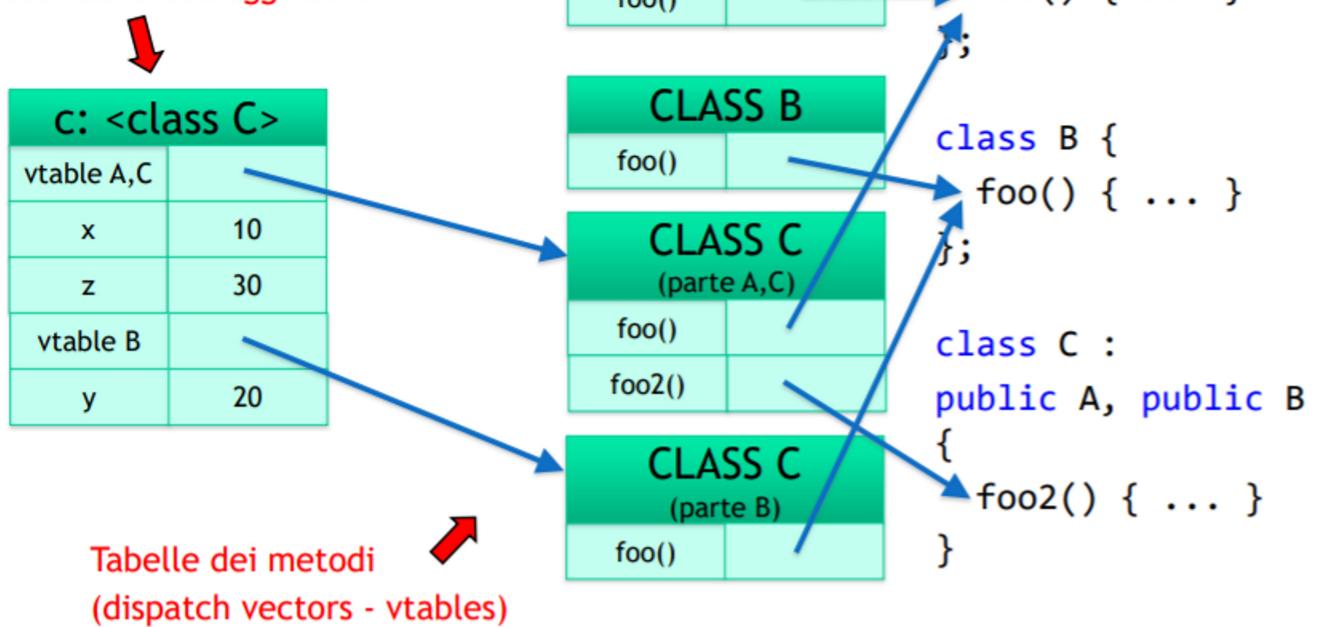
int main() {
    C c = C();
}

```

Creare un oggetto C crea prima A poi B poi C  
(stampa A, stampo B e stampo C)

## Nel runtime di C++

Descrittore dell'oggetto c



Ho nello heap l'oggetto con tutte le sue variabili di istanza, in questo caso x y e z ereditate dalle altre, ho i dispatch vector (per ogni metodo mi dice dove trovo il suo codice)

Per la classe C, nel descrittore dell'oggetto c ho, se C avesse esteso solo A avrei avuto quello con `foo()` in comune con A e poi aggiunge i metodi di B, la prima parte del descrittore ricalca quello che farei nell'eredità singola, ricopia il dispatch vector della superclasse colpuntatore al codice della superclasse

Nella seconda parte del descrittore c'è quello che succede se ereditassi solo B.

L'oggetto è proprio costruito così separato, prima ereditarietà singola poi le altre le aggiunge dopo.

In questa situazione può raggiungere entrambe le implementazioni di `foo`

Se chiamo `foo` il compilatore si lamenta

```

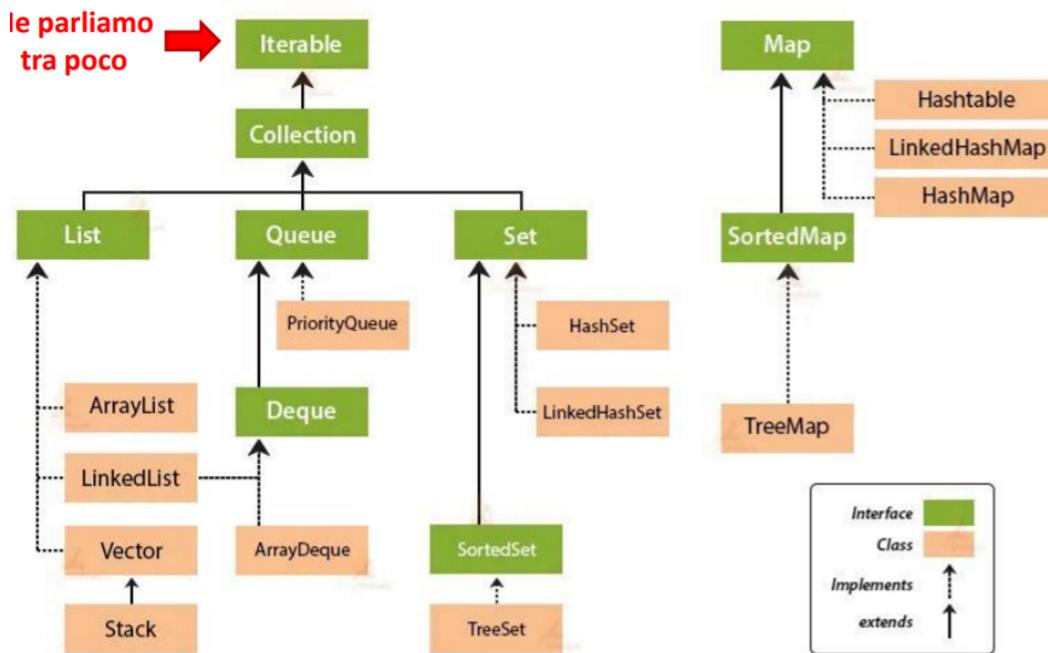
int main() {
    C c = C();
    c.A::foo() // DISAMBIGUAZIONE
}

```

In questo modo si discrimina quale dei due chiamare



## Collection Framework Hierarchy in Java



## Implementazione iteratori: IntSet

```

public class IntSet implements Iterable<Integer> {

    private int[] a;
    private int size;

    public IntSet(int capacity) { ... }
    public boolean add(int elem) throws FullSetException { ... }
    public boolean contains(int elem) { ... }

    public Iterator<Integer> iterator( ) {
        return new IntSetIterator();
    }

    // INNER CLASS (CLASSE INTERNA)
    private class IntSetIterator implements Iterator<Integer> {
        ... (SEGUE) ...
    }
}
  
```

```

public class IntSet implements Iterable<Integer> {

    ...

    // INNER CLASS (CLASSE INTERNA)
    private class IntSetIterator implements Iterator<Integer> {
        private int curr=0;

        public boolean hasNext() { return (curr<size); }

        public Integer next() {
            if (curr>=size) throw new NoSuchElementException();
            return a[curr++];
        }

        public void remove( ) {
            throw new UnsupportedOperationException( );
        }
    }
}

```

Ieratore è una classe che restituisce elementi di una collezione, un iteratore che ritorna elementi non di una collezione è un generatore (es random)

```

public class Primes implements Iterable<Integer> {
    public Iterator<Integer> iterator( )
        // EFFECTS: ritorna un generatore che produrrà' tutti
        // i numeri primi (come Integers), ciascuno una
        // sola volta, in ordine crescente
}

```

Come uso gli iteratori:

```

public class Primes implements Iterable<Integer> {
    public Iterator<Integer> iterator( );
    // EFFECTS: ritorna un iteratore che produrrà tutti i numeri
    // primi (come Integers) ciascuno una sola volta, in ordine
    // crescente
}

public static void printPrimes (int m) {
    // EFFECTS: stampa tutti i numeri primi minori o uguali a m
    // su System.out
    for (Integer p : new Primes( )) {
        if (p > m) return; // forza la terminazione
        System.out.println("The next prime is: " + p);
    }
}

```

Creo un oggetto della classe primes e poi faccio un for each su quell'oggetto

Questo foreach usa l'iteratore e tutte le volte che mi da un p va a generare un numero primo  
 Primes non è una collezione

```

public class Primes implements Iterable<Integer> {
    public Iterator<Integer> iterator( ) { return new PrimeGen( ); }
    // EFFECTS: ritorna un generatore che produrrà tutti i numeri primi
    // (come Integers) ciascuno una sola volta, in ordine crescente
    private static class PrimeGen implements Iterator<Integer> {
        // class interna statica
        private List<Integer> ps; // primi già dati
        private int p; // prossimo candidato alla generazione
        PrimeGen( ) { p = 2; ps = new ArrayList<Integer>(); } // costruttore
        public boolean hasNext( ) { return true; }
        public Integer next( ) {
            if (p == 2) { p = 3; ps.add(2); return new Integer(2); }
            for (int n = p; true; n = n + 2)
                for (int i = 0; i < ps.size( ); i++) {
                    int e1 = ps.get(i);
                    if (n%e1 == 0) break; // non è primo
                    if (e1*e1 > n) { ps.add(n); p = n + 2; return n; }
                }
        }
        public void remove( ) { throw new UnsupportedOperationException( ); }
    }
}

```

Se il numero è divisibile per quelli presenti nell'array allora non è primo

Sennò è primo

Clean

```

public void printSet (IntSet set) {
    for (Integer i : set) System.out.println(i);
}
enhanced for (for-each) su IntSet

```

----- ASTRAE DA -----

hasNext(), next(), ... definizione di un iteratore

----- ASTRAE DA -----

int[] a; int size; rappresentazione dell'insieme

----- ASTRAE DA -----

heap, garbage collection, ... gestione della memoria nella JVM

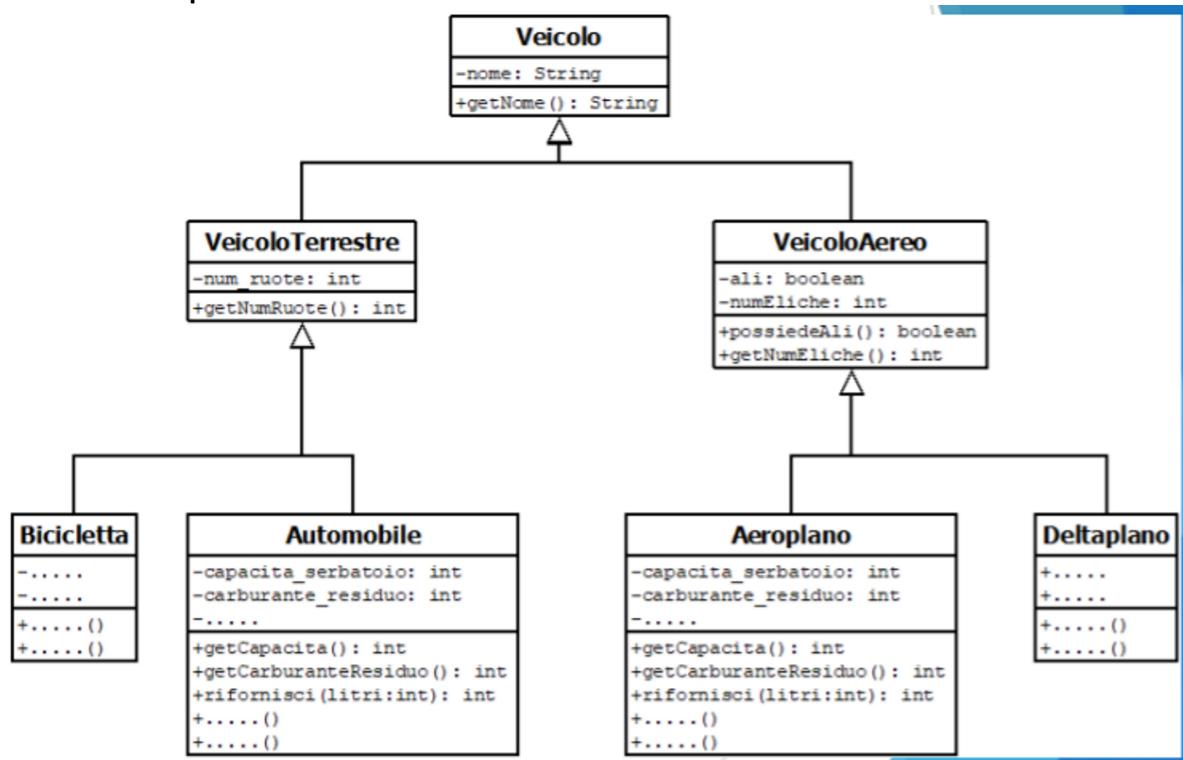
----- ASTRAE DA -----

malloc/new, free/delete, ... implementazione (in C++) della JVM

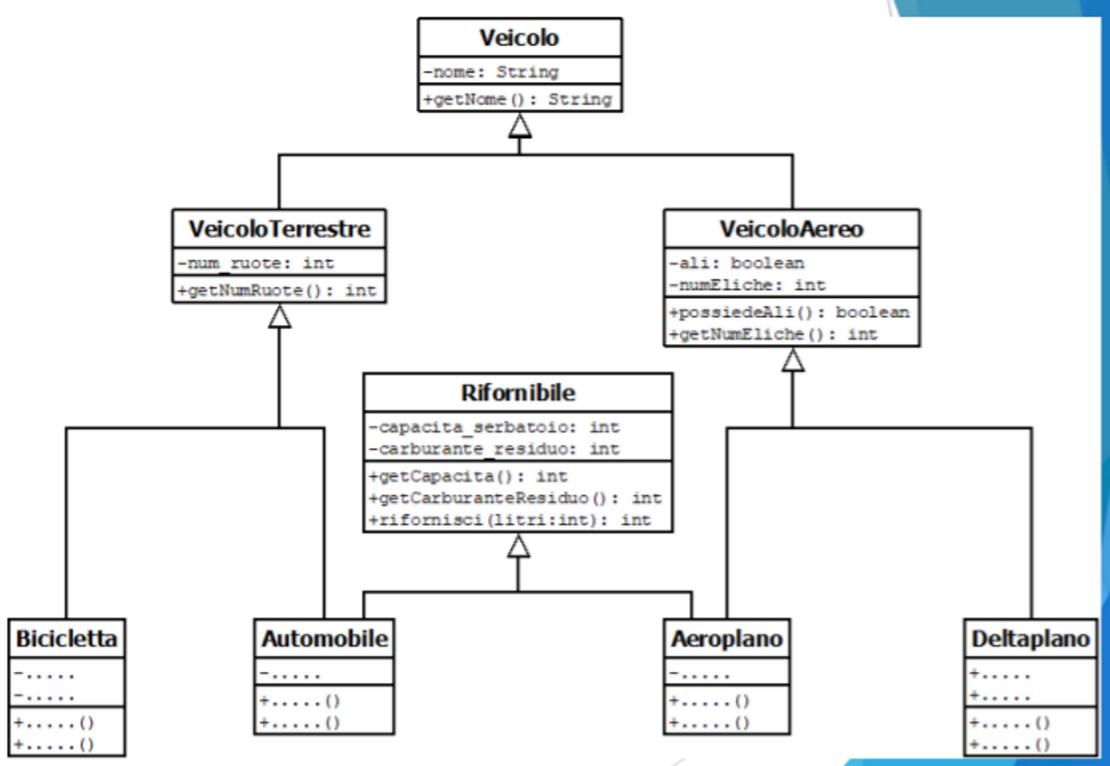
Scritto su programmazione funzionale

All'orale chiede i concetti di java che sono stati spiegati

### Ereditarietà multipla



In java non ho modo di mettere questo codice relativo a questi metodi in un unico posto, devo implementare due volte, ho del codice duplicato, posso metterci un'interfaccia ma ho comunque duplicazione del codice, mi servirebbe una superclasse comune a automobile e aereoplano -> rifornibile

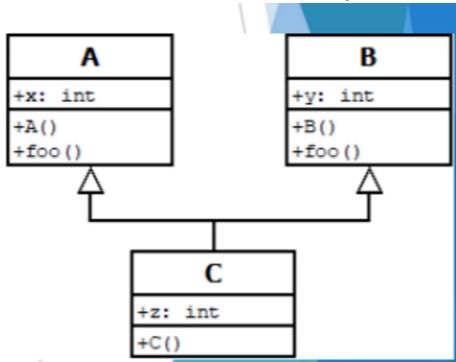


Non posso definirla come interfaccia sennò comunque devo ri-implementare il codice sia di qui che di qua

L'unica è l'ereditarietà multipla

*Perché l'ereditarietà multipla è un problema?*

Possibilità di ereditare due implementazioni diverse dello stesso metodo, quindi sono in conflitto



Dove è disponibile estendere più classi:

- C++ -> consente di disambiguare quale metodo implementare
- Java -> interfacce per definire supertipi senza definire superclassi
- Python -> gerarchia di classi come lista di classi (controllo statico)
- Dart -> composizione di classi (mixin) al posto di ereditarietà (mixin costrutto nuovo neanche troppo ben definito)

*(gli esempi sono tutti eseguibili su repl.it)*

### Esempio ereditarietà multipla in C++

```

#include <iostream>
using namespace std;

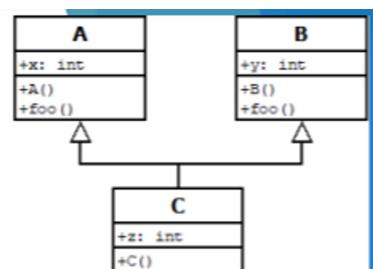
class A {
    int x = 10;
public:
    A() { cout << "A" << endl; }
    void foo() { cout << "foo A" << endl; }
};

class B {
  
```

C++

```

int
C
}
  
```



```

int main() {
    C c = C();
}
  
```

```

};

class B {
    int y = 20;
public:
    B() { cout << "B" << endl; }
    void foo() { cout << "foo B" << endl; }
};

class C : public A, public B { // estende A e B
    int z = 30;
public:
    C() { cout << "C" << endl; }
    void foo2() { cout << "foo2 C" << endl; }
}

```

```
C c = C();
```

```
}
```

Creare un oggetto C crea prima A poi B poi C  
(stampa A, stampo B e stampo C)

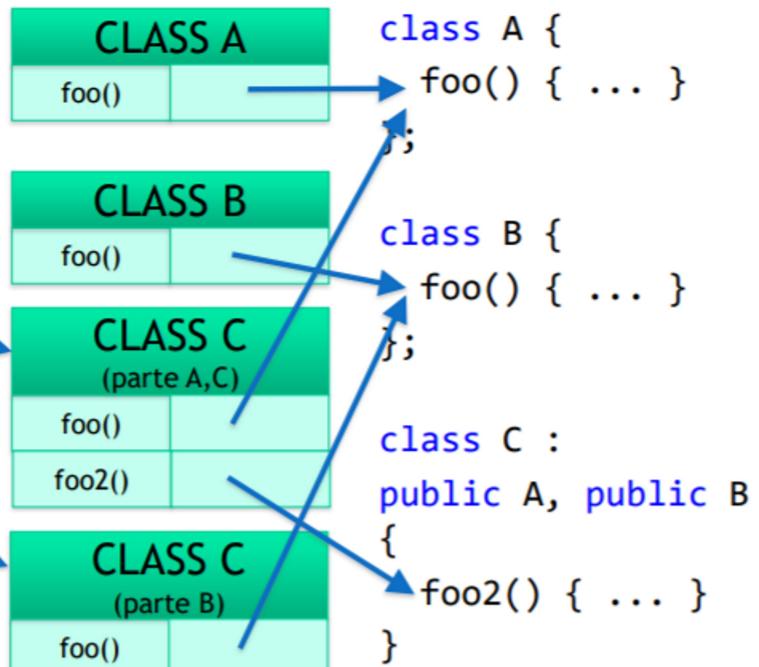
## Nel runtime di C++

Descrittore dell'oggetto c



C: <class C>	
vtable A,C	
x	10
z	30
vtable B	
y	20

Tabelle dei metodi  
(dispatch vectors - vtables)



Ho nello heap l'oggetto con tutte le sue variabili di istanza, in questo caso x y e z ereditate dalle altre, ho i dispatch vector (per ogni metodo mi dice dove trovo il suo codice)

Per la classe C, nel descrittore dell'oggetto c ho, se C avesse esteso solo A avrei avuto quello con foo() in comune con A e poi aggiunge i metodi di foo, la prima parte del descrittore ricalca quello che farei nell'eredità singola, ricopia il dispatch vectore della superclasse colpuntatore al codice della superclasse

Nella seconda parte del descrittore c'è quello che succede se ereditassi solo B.

L'oggetto è proprio costruito così separato, prima ereditarietà singola poi le altre le aggiunge dopo.  
In questa situazione può raggiungere entrambe le implementazioni di foo

Se chiamo foo il compilatore si lamenta

```
int main() {
    C c = C();
    c.A::foo() // DISAMBIGUAZIONE
}
```

In questo modo si discrimina quale dei due chiamare

```
> g++ -o main main.cpp
main.cpp: In function ‘int main()’:
main.cpp:24:5: error: request for member ‘foo’ is ambiguous
    c.foo();
    ^
main.cpp:13:10: note: candidates are: void B::foo()
                  void foo() { cout << "foo" << endl; }
```

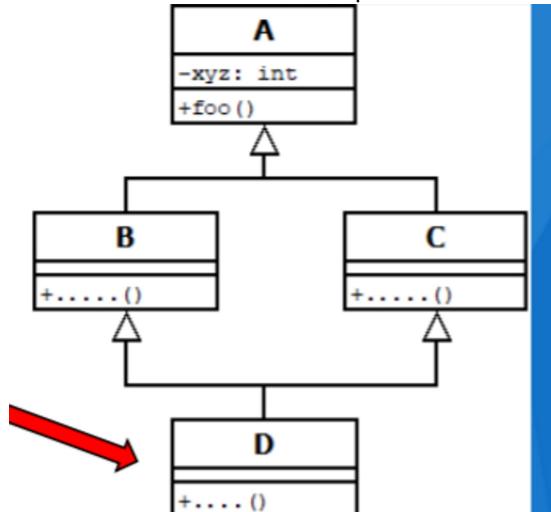
```
main.cpp:13:10: note: candidates are: void B::foo()
    void foo() { cout << "foo" << endl; }
        ^~~
main.cpp:7:10: note:                     void A::foo()
    void foo() { cout << "foo" << endl; }
        ^~~
```

19/11/2021

venerdì 19 novembre 2021 11:05

### Diamond problem

Ereditare da due superclassi che possono a loro volta avere una superclasse comune può portare a variabili d'istanza e metodi duplicati



```
#include <iostream>
using namespace std;

class A {
public:
    int xyz = 10;
    A() { cout << "A" << endl; }
    void foo() { cout << "foo A" << endl; }
};

class B : public A {
public:
    B() { cout << "B" << endl; }
};

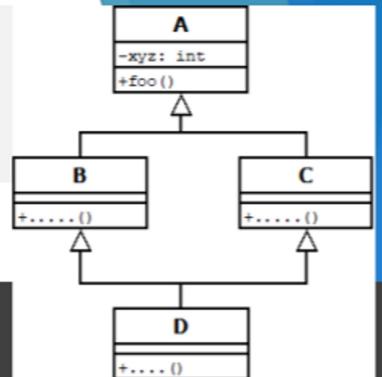
class C : public A {
public:
    C() { cout << "C" << endl; }
};

class D : public B, public C {
public:
    D() { cout << "D" << endl; }
};
```

C++

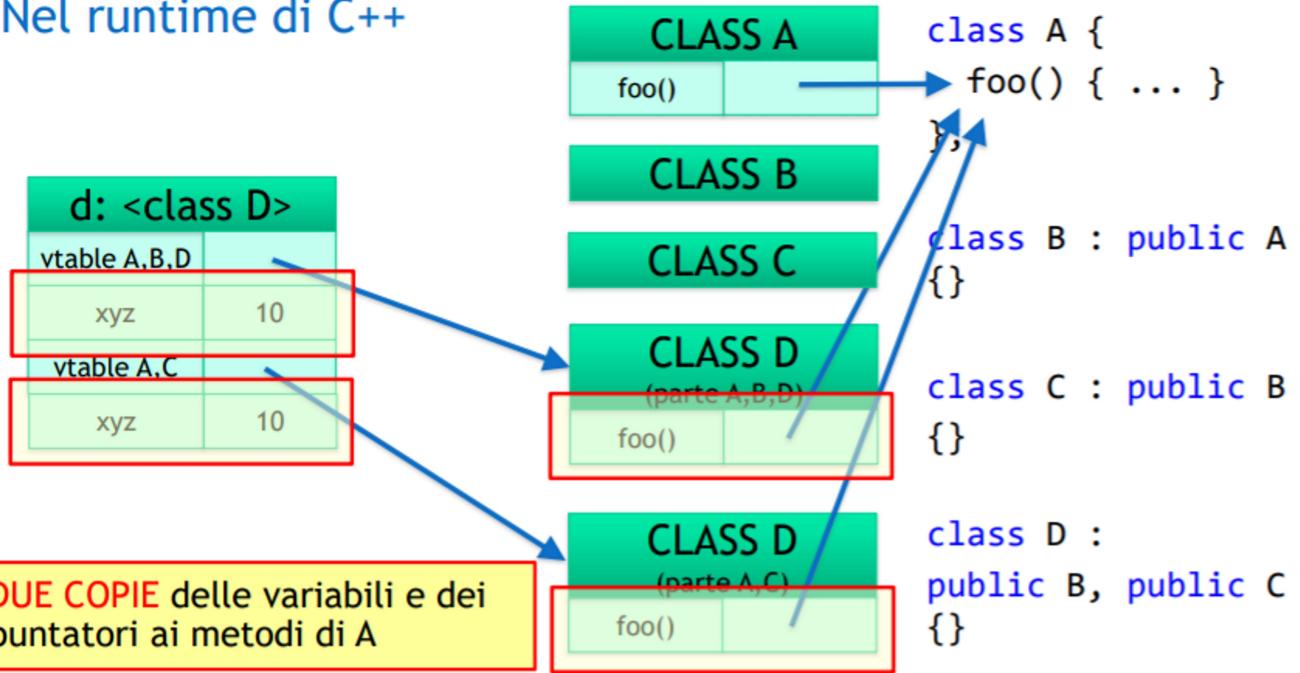
```
int main() {
    D d = D();
}
```

```
> g++ -o main main.cpp
> ./main
A
B
A
C
D
```



Il costruttore di A viene chiamato 2 volte per forza  
Da d chiama b e c che a loro volta chiamano entrambi a

## Nel runtime di C++



Sono due copie dello stesso metodo (una ereditata da b e una da A) ma le vede come due metodi fra cui dover scegliere

```

> g++ -o main main.cpp
main.cpp: In function ‘int main()’:
main.cpp:28:5: error: request for member ‘foo’ is ambiguous
    d.foo();
          ^
main.cpp:8:10: note: candidates are: void A::foo()
    void foo() { cout << "foo A" << endl; }
          ^
main.cpp:8:10: note:                     void A::foo()

```

```

class B : virtual public A {
public:
    B() { cout << "B" << endl; }
};

class C : virtual public A {
public:
    C() { cout << "C" << endl; }
};

```

La soluzione è usare **virtual**:  
 "estendo questa superclasse, ma se la estendono altre crea questa una volta sola"  
 In questo modo crea solo una volta un'istanza di A, devo mettere i virtual nei punti della gerarchia che mi possono causare più copie istanziate della stessa classe

```

> g++ -o main main.cpp
> ./main
A
B
C
D
foo A

```

Non si usa **virtual** di default per risparmiare tempo, per il numero di accessi che fa per controllare la vtable

### Ereditarietà in java

In java c'è l'ereditarietà singola, viene gestito come una semplificazione di c++ (prende solo la prima parte della tabella) java fa questa scelta per semplificare il runtime.

Per dare un'apparenza di ereditarietà multipla si implementano interfacce, ma se implementi interfacce non erediti nulla, hai solo supertipi multipli

## Interfacce e default methods

In java 8 hanno aggiunto la possibilità di utilizzare programmazione funzionale

Le collezioni hanno dovuto essere estese per essere usate anche in questo modo.

Certe interfacce sono state estese e prendono metodi in più con funzioni anonime ecc.

I programmi che utilizzavano quelle interfacce così non funzionano più, in quanto non implementano più il metodo dell'interfaccia.

Hanno introdotto delle implementazioni di *default* nelle interfacce per risolvere il problema.

-> le interfacce non sono più interfacce, hanno dei metodi implementati -> posso implementare più interfacce -> una sorta di ereditarietà multipla -> posso ereditare lo stesso metodo da più interfacce

-> *default sconsigliato*

Sia c++ che java per l'ereditarietà multipla fanno lavorare molto il compilatore

**Controllare staticamente** che le chiamate dei metodi non siano ambigue

**Generazione** delle strutture dati del runtime (dv/vtable e itable)

## I linguaggi senza compilazione?

Devono visitare la gerarchia di classe -> le chiamate di metodo sono più lente

La gerarchia di classi la descrivo con un grafo che devo visitare per cercare il codice che devo andare a eseguire

A seconda di come decido di visitare il grafo potrei arrivare a diverse implementazioni dello stesso metodo da eseguire.

Come distinguere quale metodo eseguire?

Viene linearizzato il grafo, a quel punto scorro la lista e la prima che trovo è quella che eseguo, il programmatore deve sapere come viene linearizzato il grafo per sapere quale verrà eseguita

## Method Resolution Order (Python)

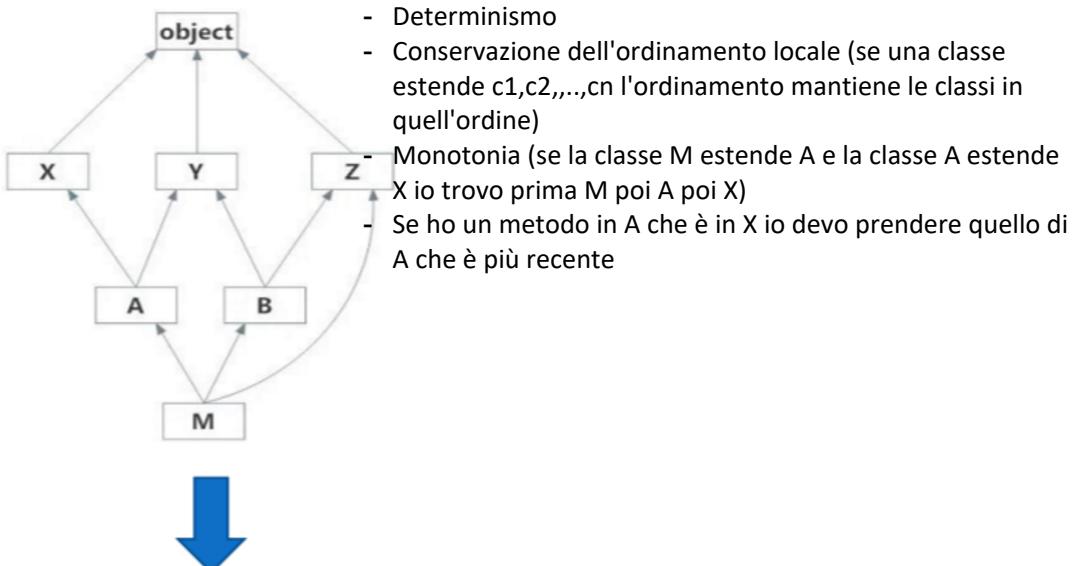
Linearizza un grafo e cerca nella lista la prima implementazione

Si basa sull'algoritmo C3 (fa una sorta di mergesort)

Soddisfa 3 proprietà:

- Determinismo
- Conservazione dell'ordinamento locale (se una classe estende c1,c2,...,cn l'ordinamento mantiene le classi in quell'ordine)
- Monotonia (se la classe M estende A e la classe A estende X io trovo prima M poi A poi X)
- Se ho un metodo in A che è in X io devo prendere quello di A che è più recente

in



[ M, A, X, B, Y, Z, object ]

In python il metodo mro() stampa la linearizzazione del grafo della gerarchia della classe su cui lo chiavi

Ci sono gerarchie di classi che non sono linearizzabili

questo da un errore a runtime

```
class A(X, Y):  
    pass
```

PYTHON

```
class A(X, Y):  
    pass  
  
class B(Y, X):  
    pass  
# non linearizzabile: ordinamenti locali  
# delle due classi incompatibili (X,Y <> Y,X)
```

PYTHON

## Allocazione dinamica: pila

### Gestione dello heap

Strutture dati che alloco con malloc (o new) strutture dati la cui nascita e morte non è legata alle chiamate di funzione.

Se faccio allocazione di oggetti all'interno di un ciclo mi si accumulano oggetti nello heap

### Come è gestito l'heap nel runtime

È un'area di memoria dove posso allocare dei pezzi per metterci strutture dati, **lista di blocchi che possono avere dimensione fissa o variabile** puntati da un puntatore iniziale che prende il nome di Lista Libera (LL)

Blocchi di dimensione fissa:

Quando alloco memoria prendo un numero di blocchi contigui che bastino per allocare quello che devo allocare

Problema: frammentazione della memoria: mi restano blocchi liberi da una porzione della lista e l'altra in cui difficilmente potrò allocare qualcosa

### Blocchi di dimensione variabile:

Parto da una lista con un unico elemento che ha come dimensione tutto l'heap, man mano che alloco vado a cercare un elemento abbastanza capiente della lista, se avanza spazio lo rимetto nella LL

- First fit: prendo il primo blocco abbastanza grande
- Best fit: prendo quello più piccolo grande abbastanza

La gestione esplicita (dal programmatore) della memoria viola il principio dell'astrazione dei linguaggi di programmazione (ci si occupa di cosa accade ad un livello più basso)

### Garbage collector

Componente del runtime che si occupa di liberare la memoria quando non viene usata

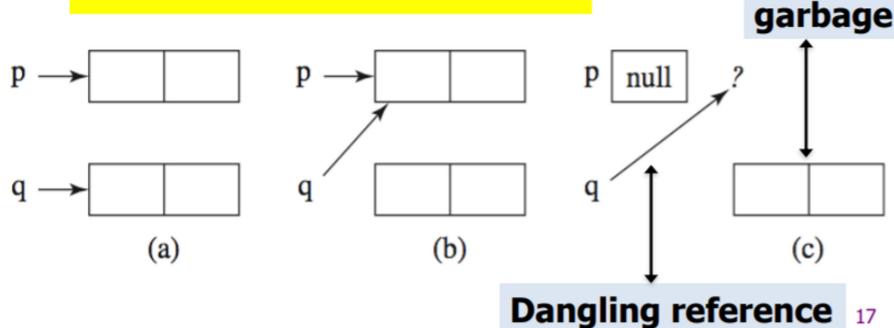
Un garbage collector efficace si occupa anche di mantenere la memoria più compatta possibile

### Come dealloca la memoria?

Deve essere in grado di capire se una porzione di memoria non è più raggiungibile dal programma (non ho più niente che punta a quella porzione di memoria)

### Garbage e dangling reference

```
class node {
    int value;
    node next;
}
node p, q;
```



## Il garbage collector **perfetto**

Nessun impatto visibile sull'esecuzione dei programmi

Opera su ogni tipo di programma e su ogni tipo di struttura dati dinamica (strutture cicliche)

Individua il garbage in modo efficiente e veloce

Nessun overhead sulla gestione della memoria complessiva

Gestione heap efficiente (nessun problema di frammentazione)

## Algoritmi di garbage collection:

### - Reference counting - contatori di riferimento

gestione diretta delle celle live

la gestione è associata alla fase di allocazione della memoria dinamica

non ha bisogno di determinare la memoria garbage

### - Tracing

identifica le celle che sono diventate garbage

- mark sweep

- copy-collection

### - Generational GC

## REFERENCE COUNTING

Aggiungere un contatore di riferimenti alle celle (numero di cammini di accesso attivi verso la cella)

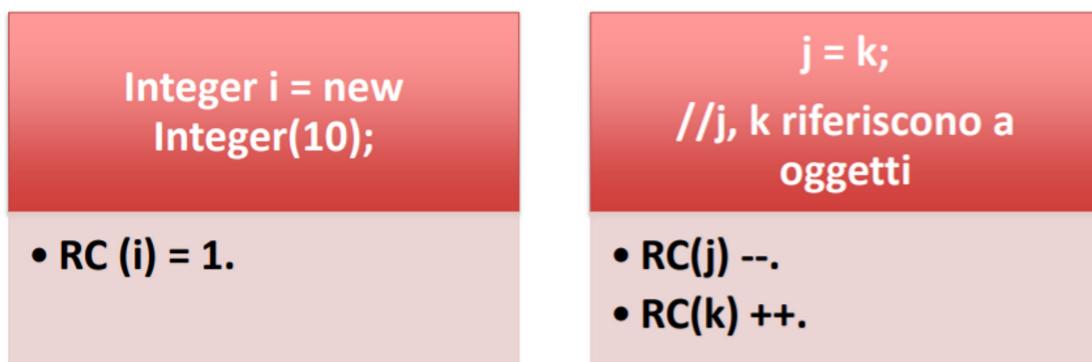
*Overhead di gestione*

- Spazio per i contatori di riferimento

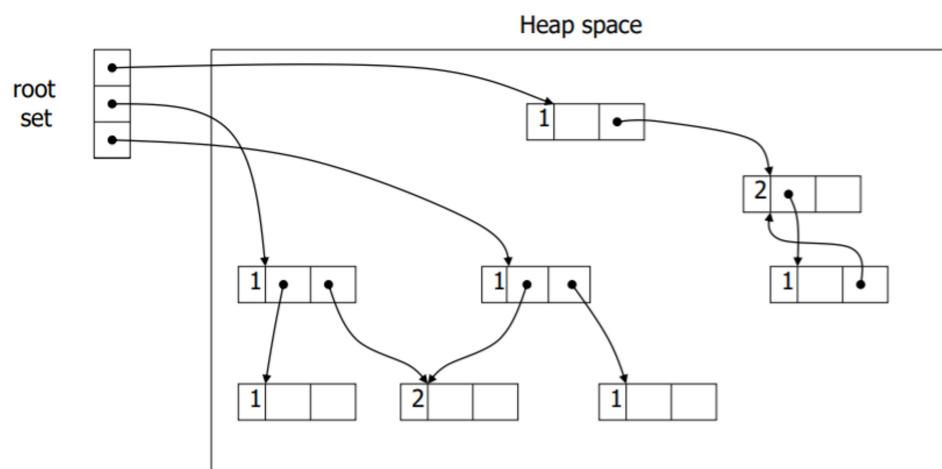
- Operazioni che modificano i puntatori richiedono incremento o decremento del valore del contatore

- Gestione "real-time"

Unix usa RC per la gestione dei file, java per la remote method invocation (RMI) c++ "smart pointer"



## Esempio



## Caratteristiche

### - Incrementale

la gestione della memoria è amalgamata direttamente con le operazioni delle primitive

linguistiche

- Facile da implementare
- Coesiste con la gestione della memoria esplicita da programma (`malloc` e `free`)
- Riuso delle celle libere immediato (se  $RC == 0$  < restituire la cella alla LL)

## Cicli

I puntatori si puntano a vicenda e quindi  $RC = 1$ , però non sono più raggiungibili dal programma



## Limitazioni

- Overhead spazio tempo  
~~memoria~~
- la modifica di un puntatore richiede diverse operazioni
- Mancata esecuzione di una operazione sul valore di  $RC$  può generare garbage
- Non permette di gestire strutture dati con cicli interni

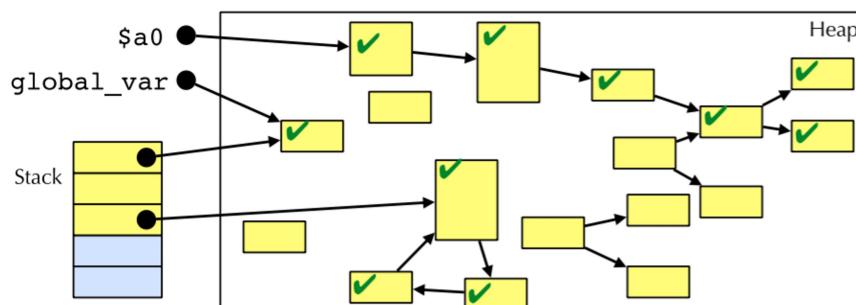
## Modello a grafo della memoria

È necessario determinare il root set: insieme dei dati "attivi" (variabili statiche + variabili allocate sul run-time stack)

Per ogni struttura dati allocata (nello stack e nello heap) occorre sapere dove ci possono essere puntatori a elementi dello heap (nei type descriptor)

**Reachable active data:** la chiusura transitiva del grafo a partire dalle radici, cioè tutti i dati raggiungibili anche indirettamente dal **root set** seguendo i puntatori

## Raggiungibilità



Come si identificano i puntatori?

Il GC, ispezionando la memoria deve essere in grado di distinguere i puntatori dalle altre cose

- Ocaml: descrittore associato alla rappresentazione (bit inferiore)
  - o 1: valore scalare (`001111` -> intero 7 non 15)
  - o 0: puntatore
  - o Sfrutta il fatto che, per via dell'accesso a parole di memoria, gli indirizzi sono valori a multipli di 4 o 8 (sempre pari, finiscono con 00)
- Java: tutti i dati (inclusi i riferimenti a oggetti) sono corredata da un esplicito descrittore con meta-dati che riportano il tipo

**Cella:** blocco di memoria sullo heap

Una cella viene detta **live** se il suo indirizzo è memorizzato in una radice o un'altra cella live -> appartiene ai reachable active data

Una cella è **garbage** se non è live

## **MARK SWEEP**

Ogni cella prevede uno spazio per un bit di marcatura  
L'attivazione del GC causa la sospensione del programma in esecuzione

### **Marking**

Si parte dal **root set** e si marcano le celle live

### **Sweep**

Tutte le celle non marcate sono garbage e sono restituite alla lista libera  
Reset del bit di marcatura

### **Valutazione**

Opera correttamente sulle strutture circolari  
Nessun overhead di spazio

Sospende l'esecuzione  
Non interviene sulla frammentazione dello heap

## **COPYING COLLECTION**

L'algoritmo di cheney è un algoritmo di GC che opera suddividendo la memoria heap in due parti:

### **From space e to space**

Solo una parte dello heap è attiva (permette di allocare nuove celle)  
Quando viene attivato il GC, le celle live vengono copiate nella seconda porzione dello heap (quella non attiva)  
Alla fine della copia i ruoli tra le due parti dello heap vengono scambiati  
Le celle nella parte non attiva vengono restituite alla LL in un unico blocco

### **Valutazione**

Efficace nell'allocazione di porzioni di spazio di dimensioni differenti e evita problemi di frammentazione

Duplicazione dello heap (funziona bene se si ha tanta memoria)

### **Approccio generazionale**

Heap diviso in aree, non intercambiabile

- Young: oggetti creati recentemente
- Old: oggetti più "anziani"

Ci possono essere anche più di 2 aree.

Avendo più aree si possono applicare algoritmi diversi su aree diverse

Su due aree si usa una sorta di copy collection

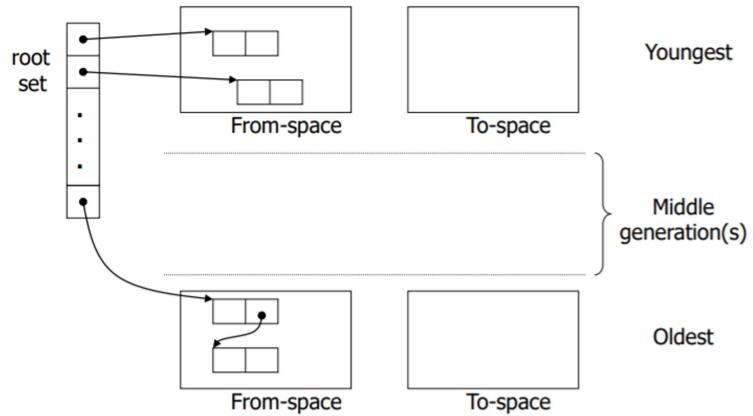
Si alloca sempre nell'area young, si copiano gli oggetti vivi in old (senza scambiare) si ripulisce young di oggetti rimasti

Nell'area old si usa un altro algoritmo (es mark & sweep) per ripulire il garbage (meno frequentemente di quanto pulsici young)

Vantaggio:

Mi consente di concentrarmi a tempi diversi su oggetti che hanno vite di durata diversa  
Rispetto al copy collection mi risparmio il tempo di doverli ricopiare da la, risparmio tempo se ho molti oggetti di durata breve

In linguaggi come java ho più fasce d'età



## GC nella pratica

- Sun/Oracle Hotspot JVM
  - GC con tre generazioni (0, 1, 2)
  - Gen. 0,1 copy collection
  - Gen. 2 mark-sweep con meccanismi per evitare la frammentazione
- Microsoft .NET
  - GC con tre generazioni (0, 1, 2)
  - Gen. 2 mark-sweep (non sempre compatta i blocchi sullo heap)

25/11/2021

giovedì 25 novembre 2021 14:55

**Programmazione concorrente**

L'interleaving ha luogo a livello del linguaggio macchina, non quello a alto livello

Anche con vero parallelismo due processi non possono scrivere contemporaneamente nella stessa  
locazione di memoria, scriveranno prima uno poi l'altro comunque -> interleaving

17/11/2021

giovedì 4 novembre 2021 13:30