

## 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 comportano 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

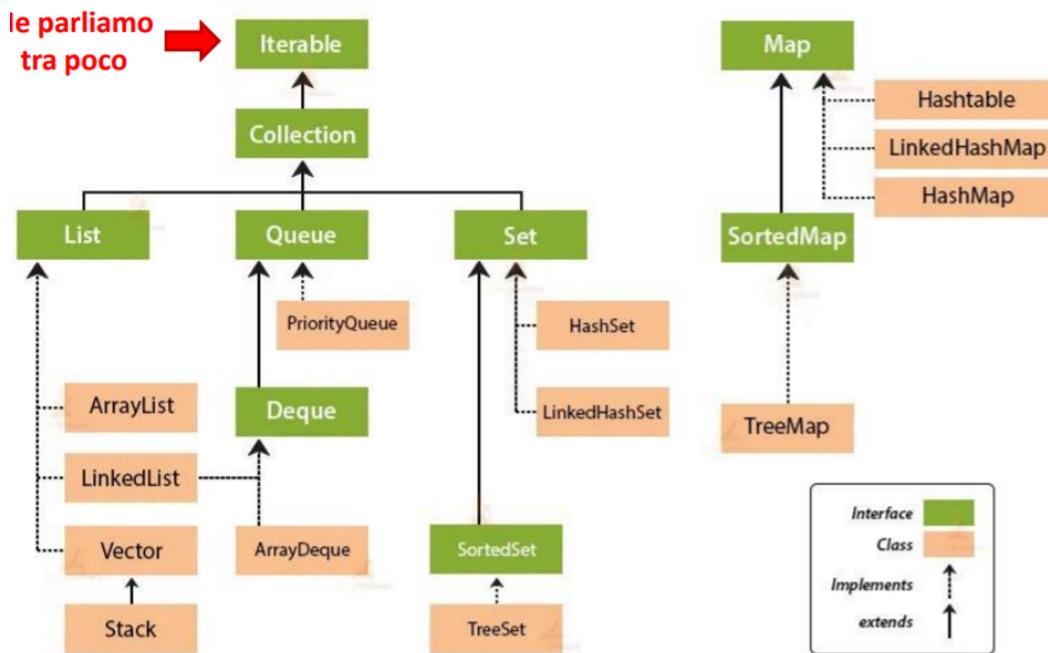
È in contrasto su quello detto prima (nelle slide).

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

- ▶ 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

Il tipo generico si perde nella traduzione al bytecode

## 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

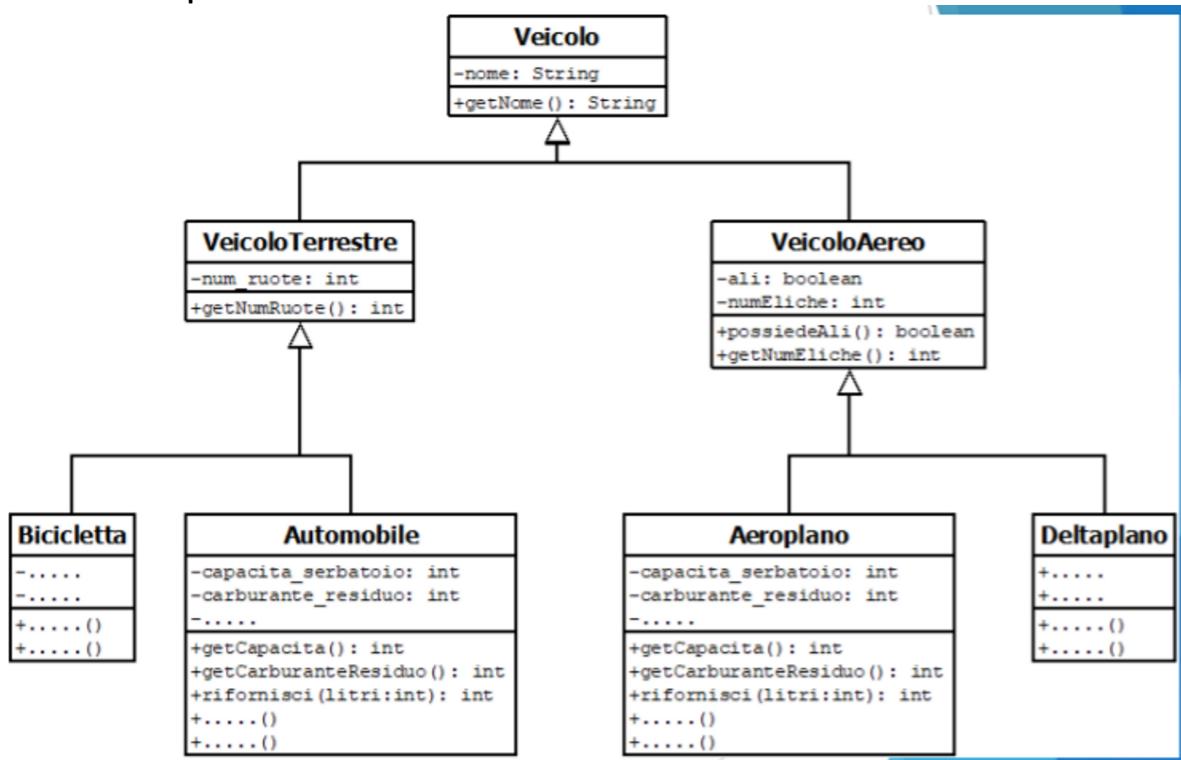
----- ASTRAE DA -----

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

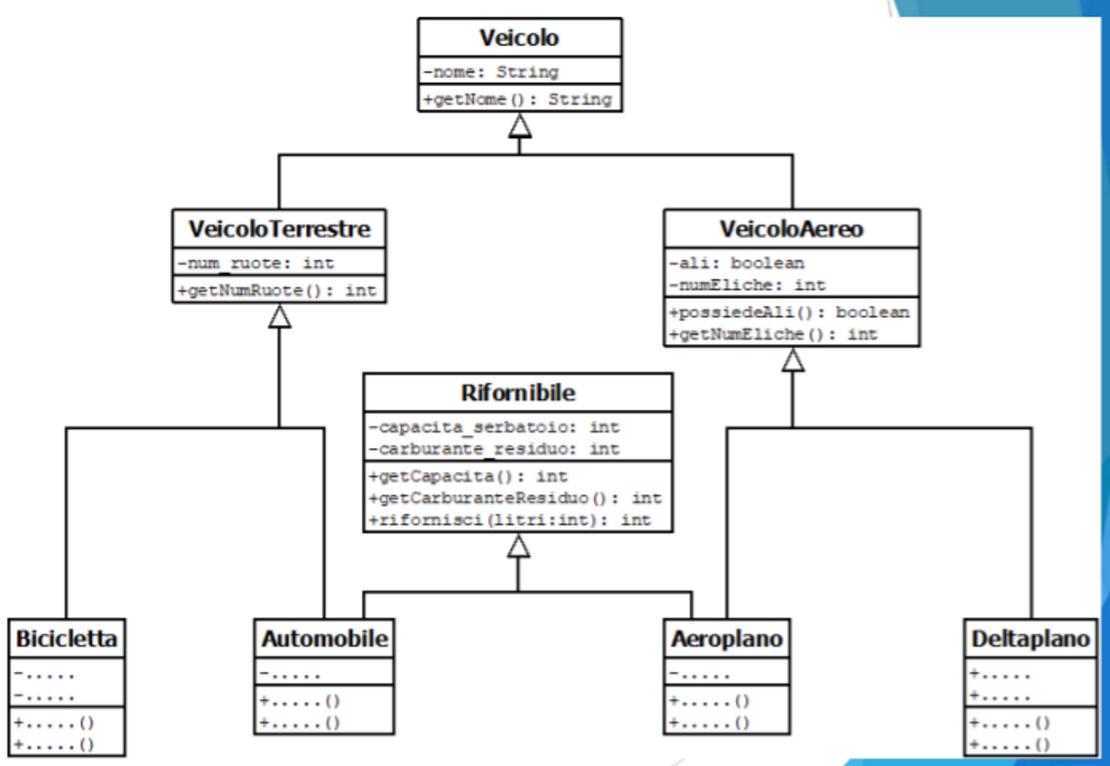
Scritto su programmazione funzionale

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

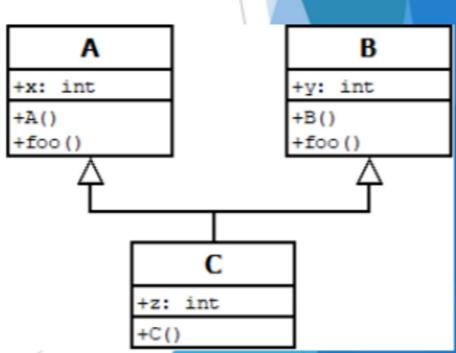


Non posso definirla come interfaccia sennò comunque devo ri-implementare il codice sia di qui che di qua

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

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

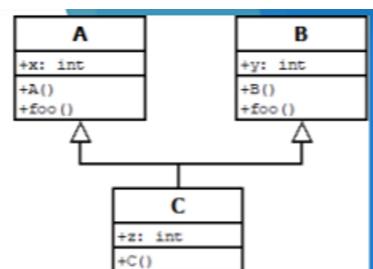
```

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)

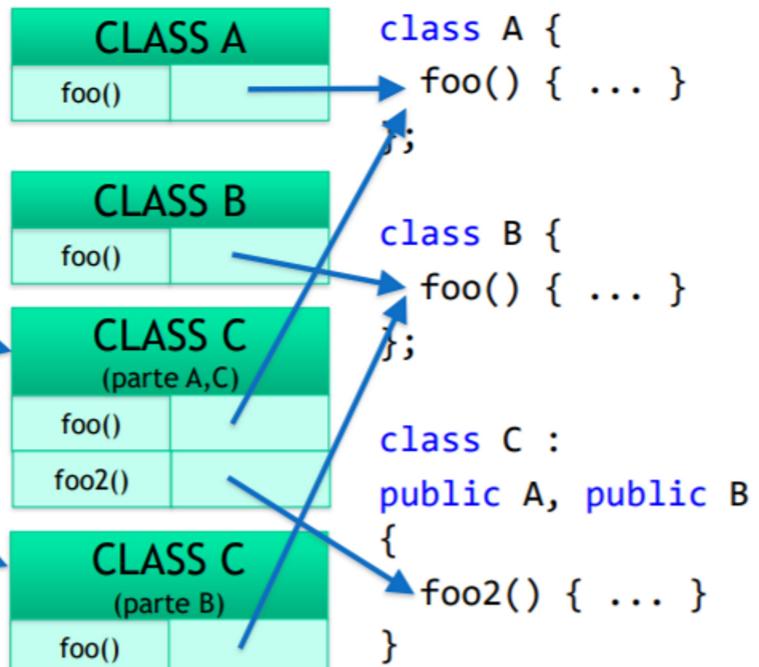
## Nel runtime di C++

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; }

```

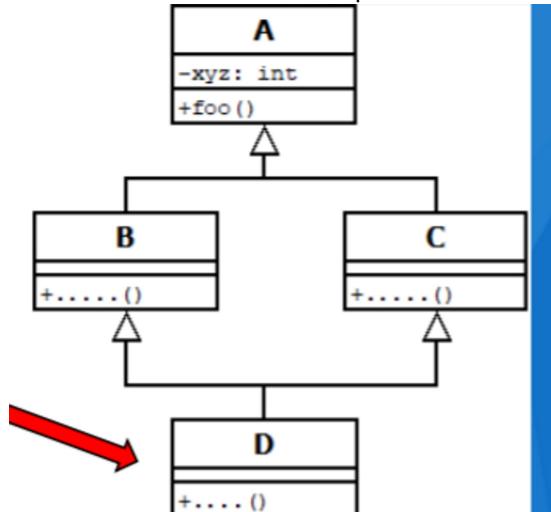
```
main.cpp:13:10: note: candidates are: void B::foo()
    void foo() { cout << "foo" << endl; }
        ^~~
main.cpp:7:10: note:                     void A::foo()
    void foo() { cout << "foo" << endl; }
        ^~~
```

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; }
};

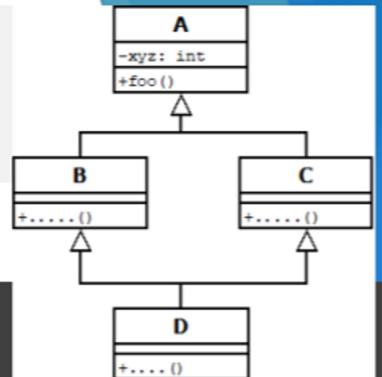
class C : public A {
public:
    C() { cout << "C" << endl; }
};

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

C++

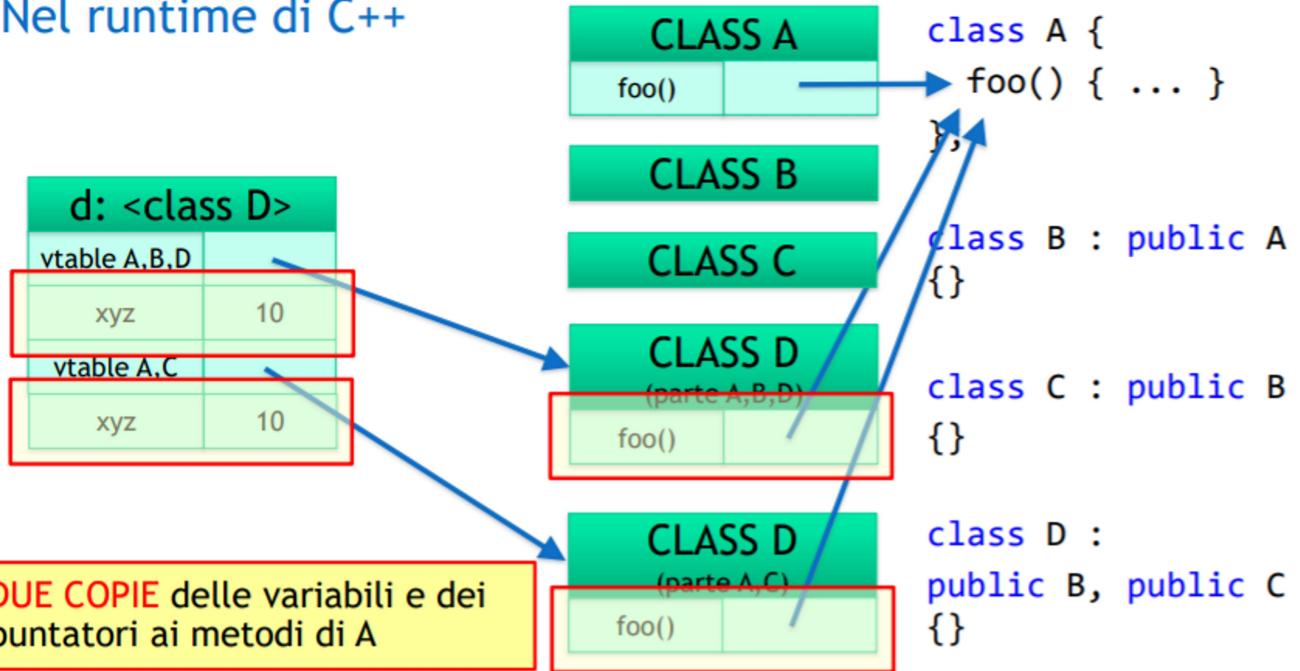
```
int main() {
    D d = D();
}
```

```
> 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)

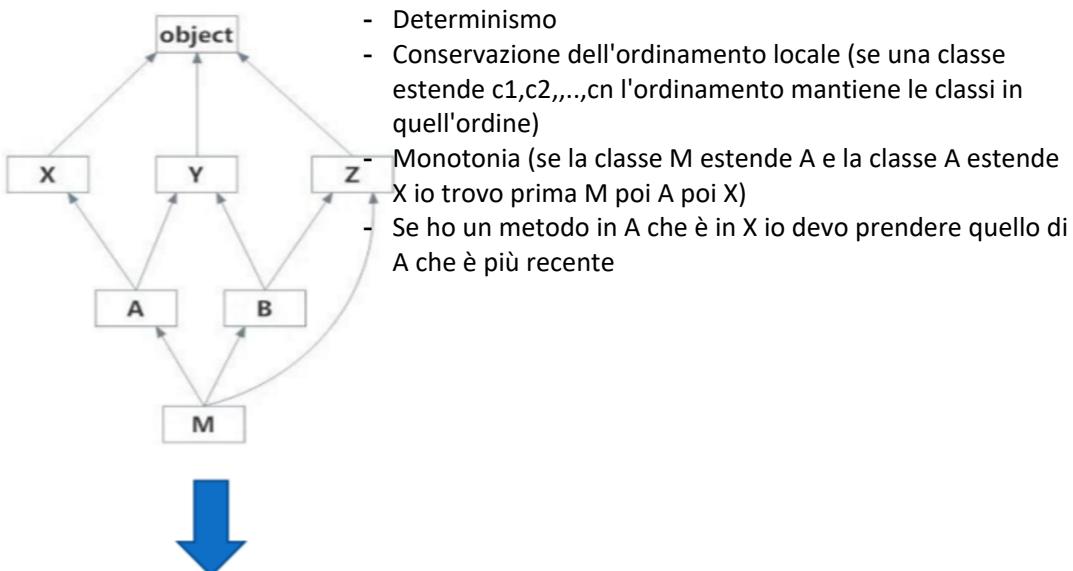
Linearizza un grafo e cerca nella lista la prima implementazione

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

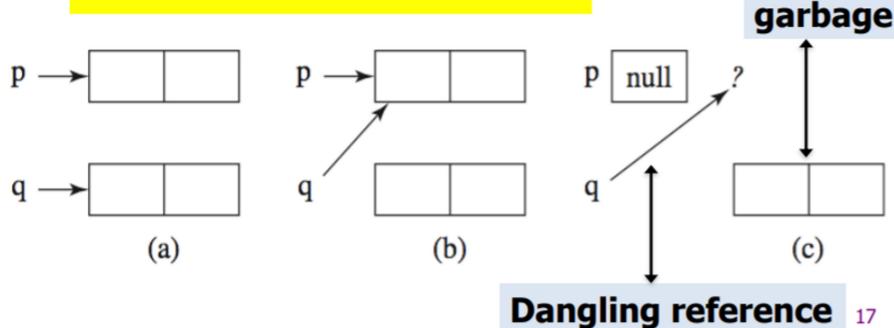
Un garbage collector efficace si occupa anche di mantenere la memoria più compatta possibile

### 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)

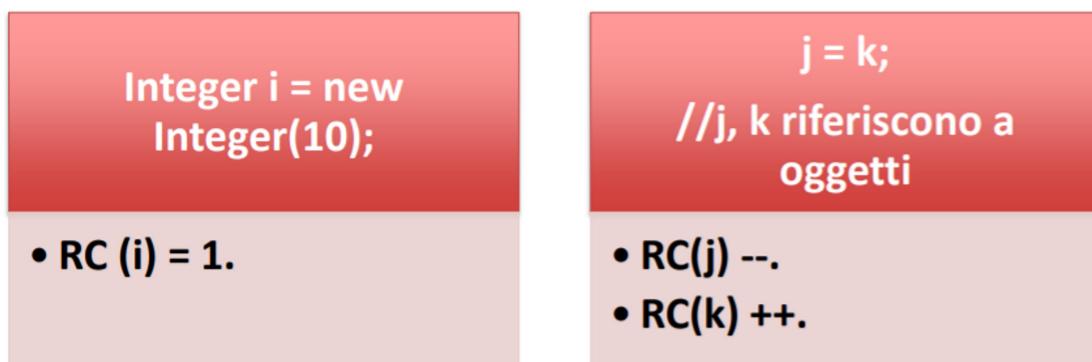
*Overhead di gestione*

- Spazio per i contatori di riferimento

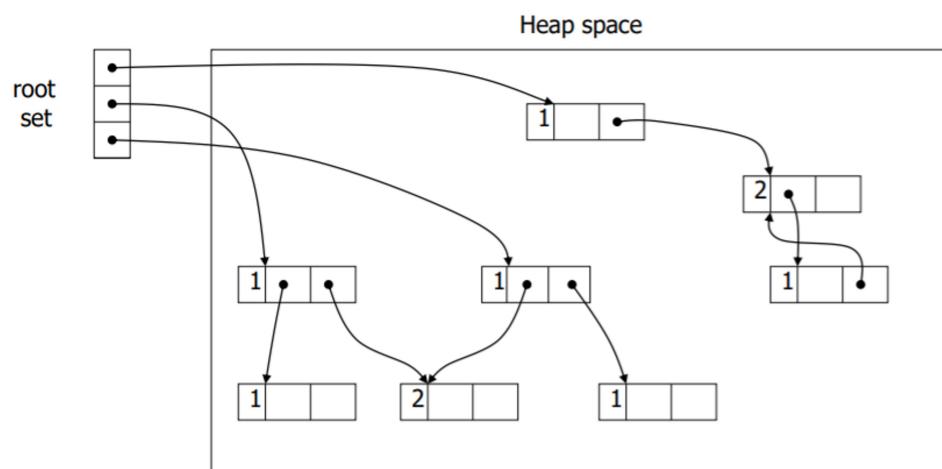
- Operazioni che modificano i puntatori richiedono incremento o decremento del valore del contatore

- 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  
spazio per il contatore  
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**

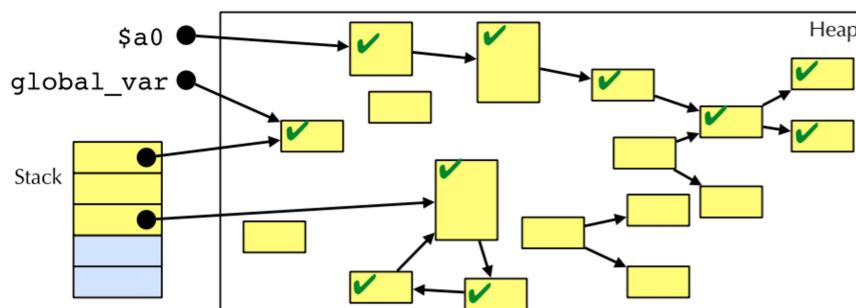
## Modello a grafo della memoria

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

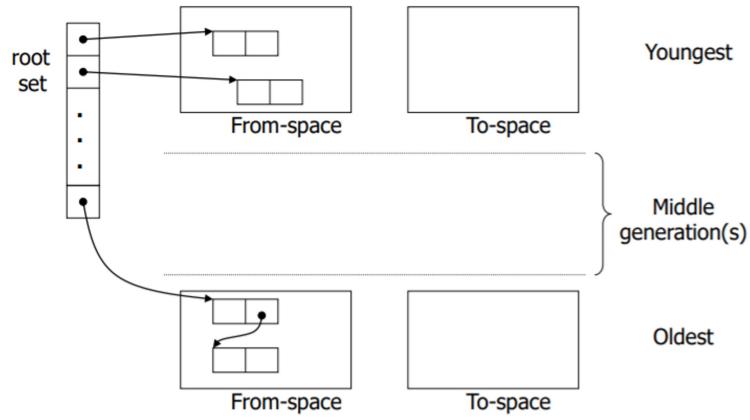
Si alloca sempre nell'area young, si copiano gli oggetti vivi in old (senza scambiare) si ripulisce young di oggetti rimasti

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 di 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