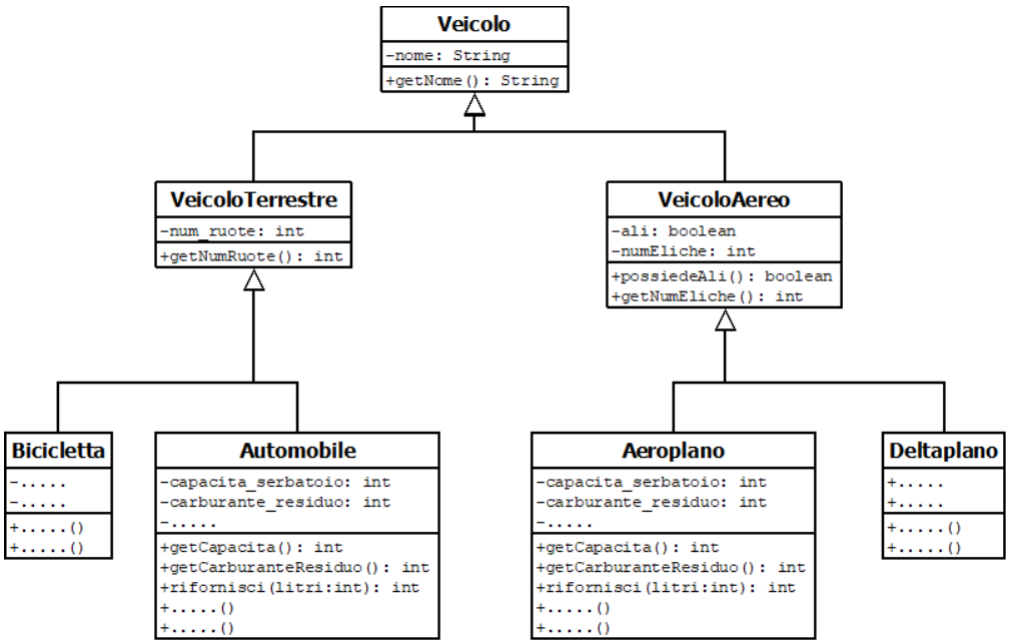
Giorno 6
OOP: Ereditarietà multipla

1 Ereditarietà Multipla

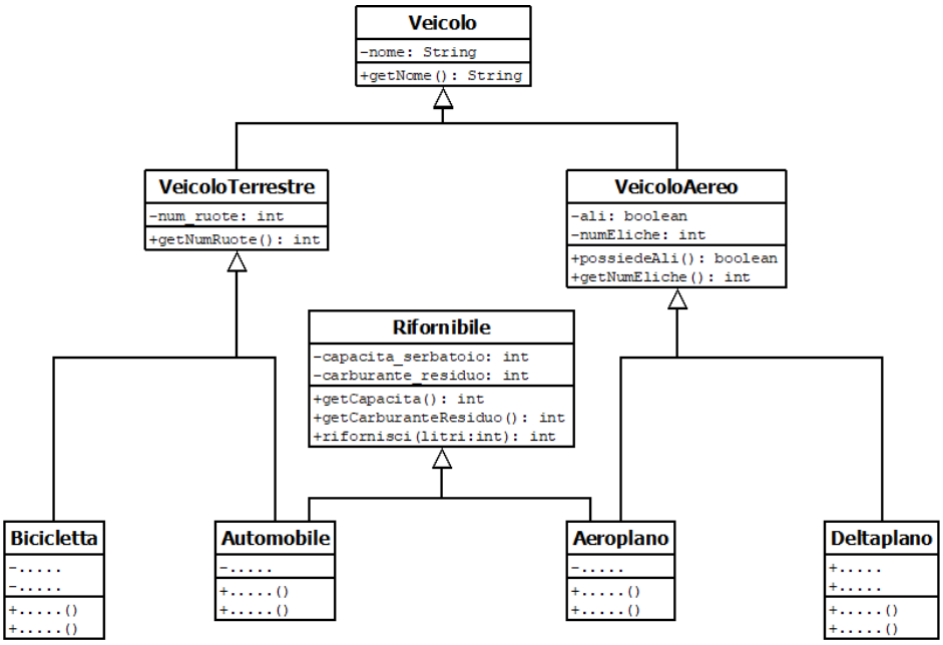
1.1 Introduzione e motivazione

Assumiamo di avere la seguente gerarchia di classi:



Le classi "Automobile" ed "Aeroplano" hanno in comune dei metodi e degli attributi che non sono presenti nelle rispettive superclassi; dobbiamo implementare il codice due volte. In Java potremmo far sì che queste due classi implementino un'interfaccia comune, ma questo non risolverebbe (dobbiamo comunque scrivere il codice dei metodi due volte...)

Vorremmo fare in modo che queste due classi estendano, oltre alle superclassi in figura, un'altra classe "Rifornibile", che fornisce i metodi e gli attributi in comune:



Questo non si può fare in Java (ma in java 8 hanno aggiunto implementazioni standard nelle interfacce...)

1.2 Problema dell'Ereditarietà Multipla

- Se entrambi i metodi estesi hanno un loro metodo con la stessa signature: quale si sceglie?
- **Diamond problem**: Ereditare da due superclassi che possono **a loro volta avere una superclasse in comune** può portare a variabili d'istanza e metodi duplicati.

1.3 Ereditarietà multipla in C++

1.3.1 Disambiguazione

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

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

- Si **noti** che non è stato scritto "new" davanti a `C()`
- Eseguire questo codice stampa `A B C`, poiché creare un oggetto C evoca i costruttori di A e B, nell'ordine, prima di eseguire il proprio.
- A livello del **runtime di C++**, Il dispatch vector di C è diviso in due parti: una con i metodi di C ed A (ossia, il dispatch vector "come se ci fosse ereditarietà singola") ed uno che punta ai metodi di B: questo ci permette di scegliere quale versione di `foo()` eseguire, usando questa sintassi:

```cpp
c.A::foo()
```

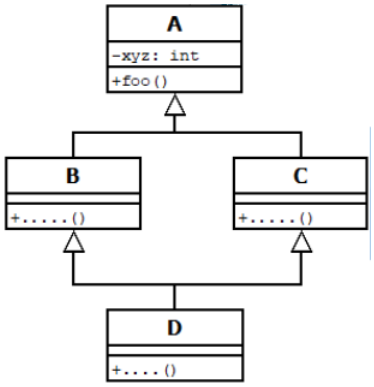
(vedi slide per schema – Trattazione non estensiva, rivedere pezzettino di lezione se necessario)

1.3.2 Diamond problem

Se due classi B e C estendono una singola classe A, ed una classe D estende B e C, che succede se cerco di chiamare un metodo di A?

C++ pensa che ci sia da disambiguare tra due istanze del metodo: Il problema si scrivendo che le classi B e C estendono **virtual** A, invece di A.

In questo modo A è generato una volta sola.



Esempio

```
#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 { // da sostituire con "virtual public A"
    public:
        B() { cout << "B" << endl; }
};
class C : public A { // da sostituire con "virtual public A"
    public:
        C() { cout << "C" << endl; }
};
class D : public B, public C {
    public:
        D() { cout << "D" << endl; }
};
```

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

Il programma stampa: `A B A C D`; il costruttore di A è chiamato due volte, e se cerco di chiamare `d.foo()` ottengo un errore "reques for member 'foo' is ambiguous", che propone come candidati i due metodi `foo` delle due istanze di A.

Nel runtime di C++, la classe D è costituita di due dispatch vectors:

- Parte A, B, D (cammino più a sinistra)
- Parte A, C (cammino a destra)

E la tabella dell'oggetto d è così strutturata

vtable A, B, D	→ parte A, B, D
xyz	10
vtable A, C	→ parte A, C
xyz	10

Soluzione: Se definisco B e C come `virtual`, il programma stampa `A B C D`, nessun problema.

Nel runtime di C++, la classe D è costituita di tre dispatch vectors:

- Parte B, D (cammino più a sinistra)
- Parte C
- Parte A (da sola, perché è virtual)

E la tabella dell'oggetto d è così strutturata

vtable B, D	→ parte B, D
A	→ ($\star$)
vtable C	→ parte C
A	→ ($\star$)
vtable A	($\star$) → parte A
xyz	10

Come si può notare, per accedere alla vtable di A c'è l'overhead di deferenziare un puntatore in più.

Perché virtual non è di default? Per risparmiare tempo, proprio a causa di quella dereference in più: queste dereference potrebbero infatti concatenarsi ed accumularsi.

1.4 Interfacce multiple in Java

In Java non c'è l'ereditarietà multipla, ma si possono definire più interfacce.

Problema Non si può usare lo sharing strutturale! Questo perché, se si implementano due interfacce, ci sono più membri con lo stesso indice (e.g. il primo elemento delle due interfacce ha indice zero)

Si usa perciò una "itable", che contiene tutte le interfacce e i puntatori dai nomi dei metodi al loro codice.

- Vantaggi: La soluzione per la gestione edlle interfacce è trasparente al programmatore
- Svantaggi: uso limitato dei meccanismi di ereditarietà

1.4.1 Default methods

Da java 8 si possono mettere implementazioni di default dei metodi nelle interfacce: questo ha introdotto di fatto l'ereditarietà multipla, con tutti i suoi problemi: **problema**, in caso di ambiguità il compilatore dà errore ed il programmatore deve modificare il codice dei metodi in conflitto (che schifo).

Perché questa aggiunta? Gli sviluppatori Java hanno aggiunto le funzioni anonime, ed hanno esteso le collezioni per supportarle. **Problema**: non potevano aggiungere metodi alle interfacce se questi sarebbero risultati non implementati in tutti i programmi che usavano le collezioni, per retro compatibilità; perciò hanno aggiunto la possibilità di avere delle implementazioni di default.

1.5 Ereditarietà multipla per linguaggi interpretati

Nei linguaggi interpretati, la gestione dell'ereditarietà è basata su una visita del grafo che rappresenta la gerarchia. Visite diverse implicano scelte diverse in caso di ambiguità.

1.5.1 Python

Python linearizza il grafo (lo porta in *method resolution order*) utilizzanodo l'algoritmo **C3**, che garantisce:

- Determinismo
- Conservazione dell'ordinamento locale (se c estende $c_1 \cdots c_n$ l'ordinamento prevede queste classi in quest'ordine)
- Monotonia (c_1 sottoclasse di $c_2 \implies c_1$ viene prima di c_2)

L'ordine si può ottenere, in python, usando `CLASSE.mro()`

Esistono gerarchie di classi non linearizzabili:

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

($\implies$ errore a tempo di esecuzione.)

1.6 Mixin

Un mixin è un componente che può essere "mescolato" ad una classe esistente. Posso usarlo per aggiungere metodi **condivisi** a più classi, senza creare una superclasse. Se si considera l'esempio iniziale di automobile ed aeroplano, "rifornibile" potrebbe essere realizzato come Mixin; in questo modo non c'è bisogno, in questi contesti, di ricorrere all'ereditarietà multipla.

Ad esempio, in **Dart**, si può modificare una classe con un mixin utilizzando **with**.