MULTIPLE INHERITANCE (E MIXINS)

Ereditarietà Multipla (Multiple Inheritance)

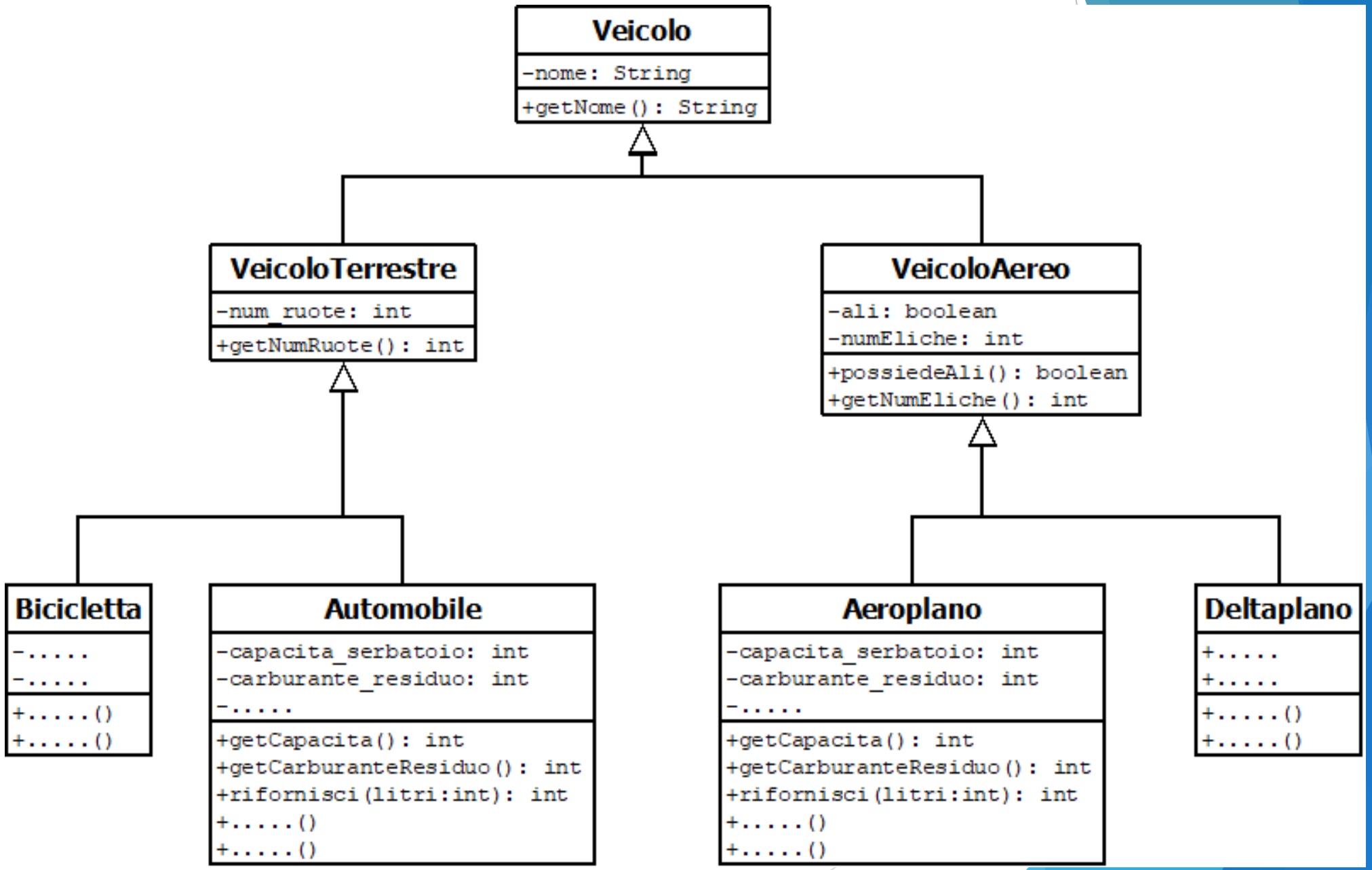
In Java è possibile definire una classe come estensione di una (singola) altra classe esistente usando la parola chiave extends

- `class B extends A {…}`
- `A` è super-classe (e da origine a un super-tipo), mentre `B` è sotto-classe (e da origine a un sotto-tipo)

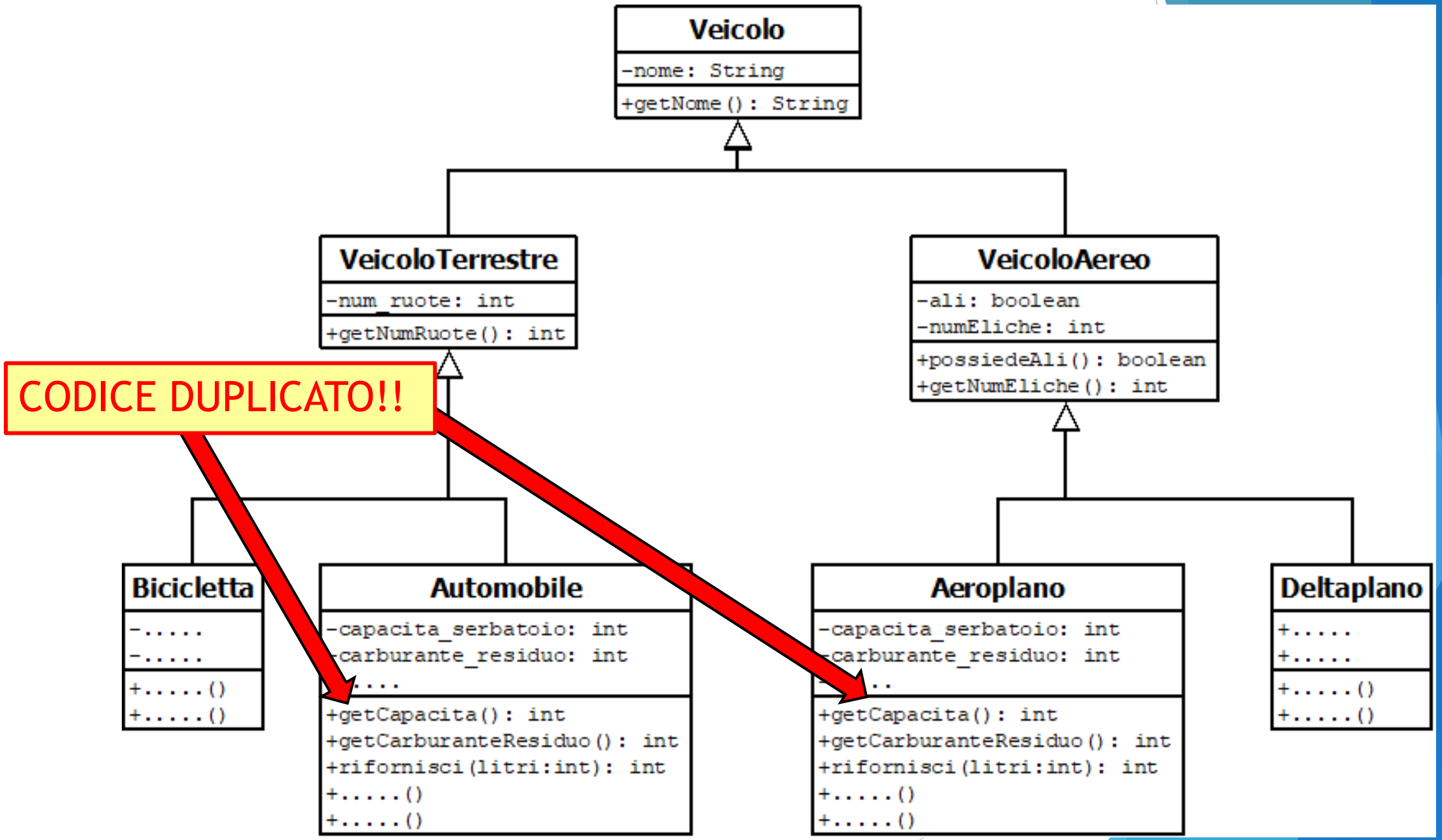
In alcune situazioni sarebbe utile avere la possibilità di estendere più classi (ereditarietà multipla)

- In alcuni linguaggi (es. C++) questo è possibile

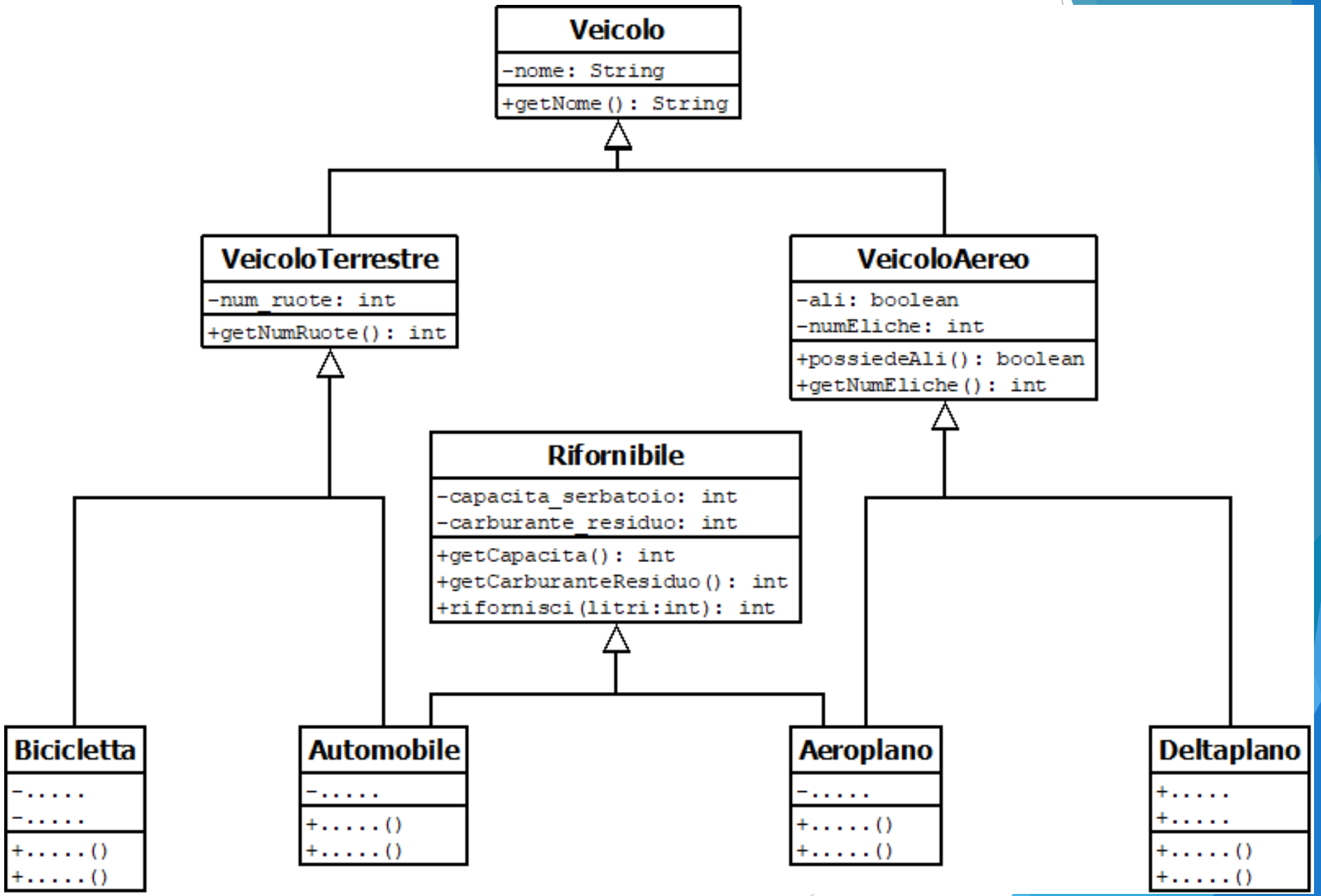
Esempio di gerarchia di classi



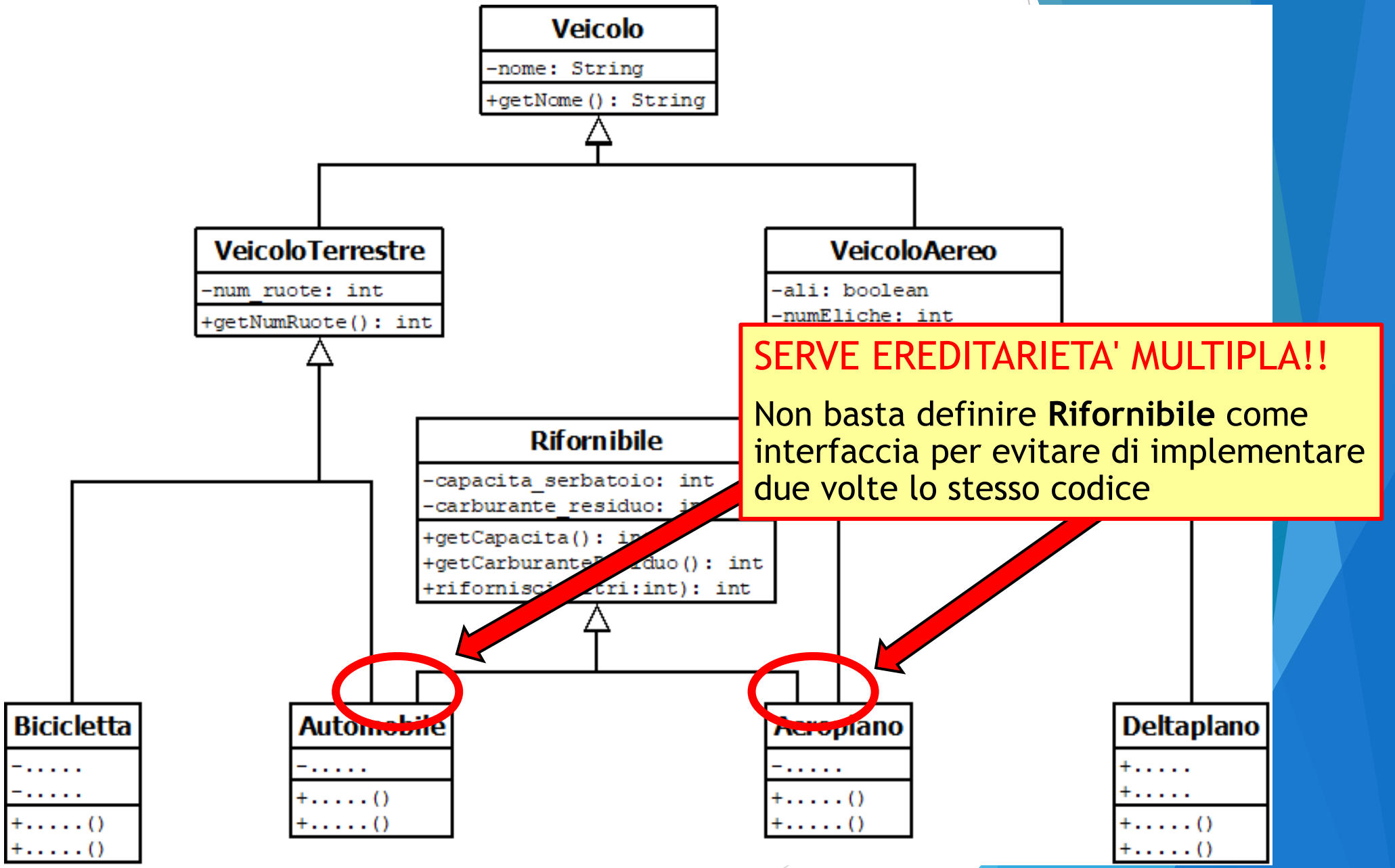
Esempio di gerarchia di classi



Esempio di gerarchia di classi con ereditarietà multipla



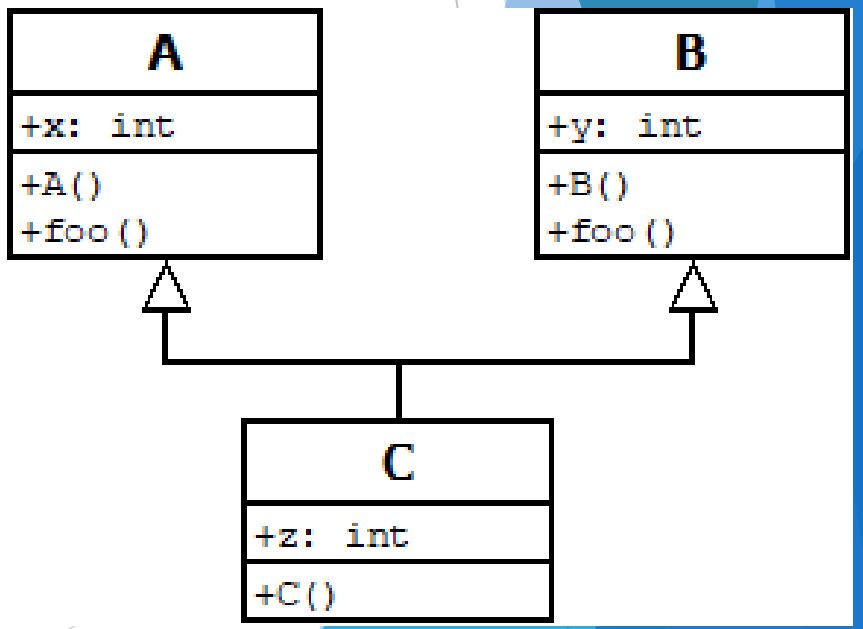
Esempio di gerarchia di classi con ereditarietà multipla



Perché l'ereditarietà multipla è un problema?

L'ereditarietà multipla apre alla possibilità di ereditare diverse implementazioni dello stesso metodo

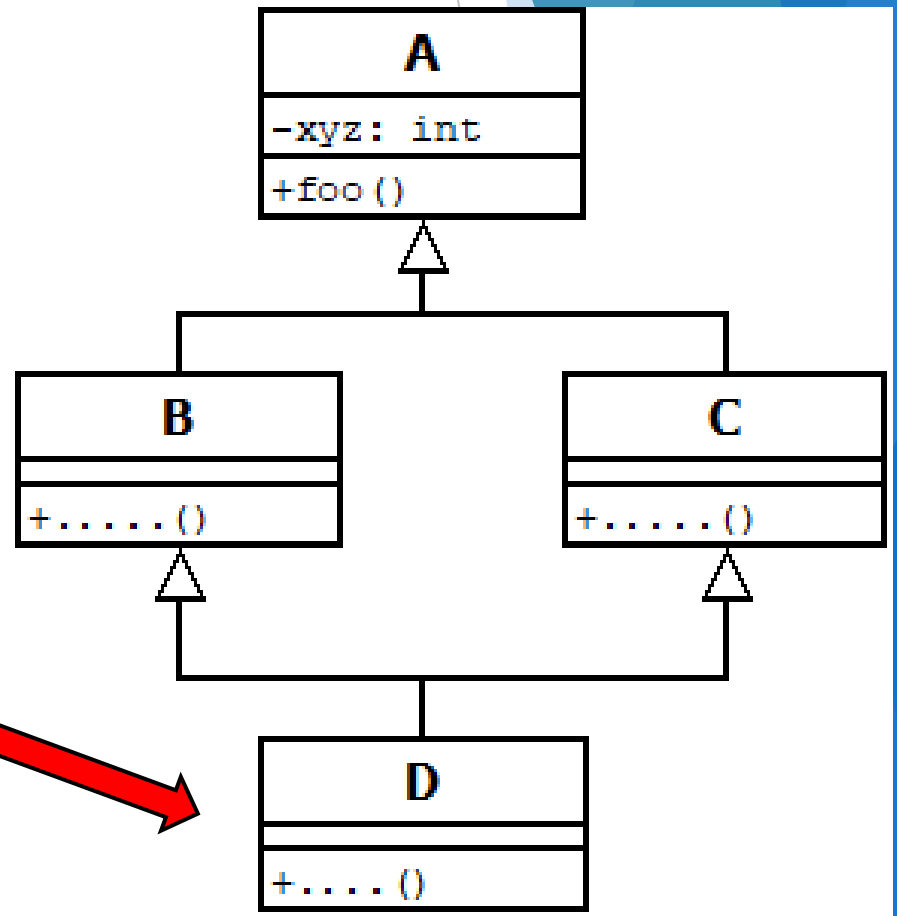
```
C obj = new C();
obj.foo(); // quale viene eseguito?
           // quello di A o di B?
```



Perché l'ereditarietà multipla è un problema?

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



Potrebbe ereditare due copie di xyz e di foo()

Multiple Inheritance nei linguaggi class-based

Diverse soluzioni disponibili

- C++: ereditarietà multipla e possibilità di disambiguare
- Java: interfacce per definire supertipi senza definire superclassi
  - Nota: da Java 8 le interfacce possono avere implementazioni di default!!

Le soluzioni di sopra richiedono controlli statici

- Python: linearizzazione della gerarchia di classi (algoritmo C3)

Cambio di prospettiva:

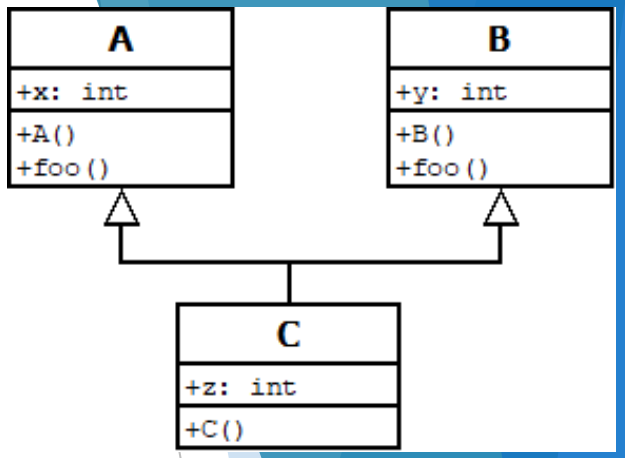
- Dart (e altri): composizione di classi (mixin) al posto di ereditarietà

Gli esempi che vedremo sui vari linguaggi sono tutti eseguibili su replit

C++

```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; }
};
```

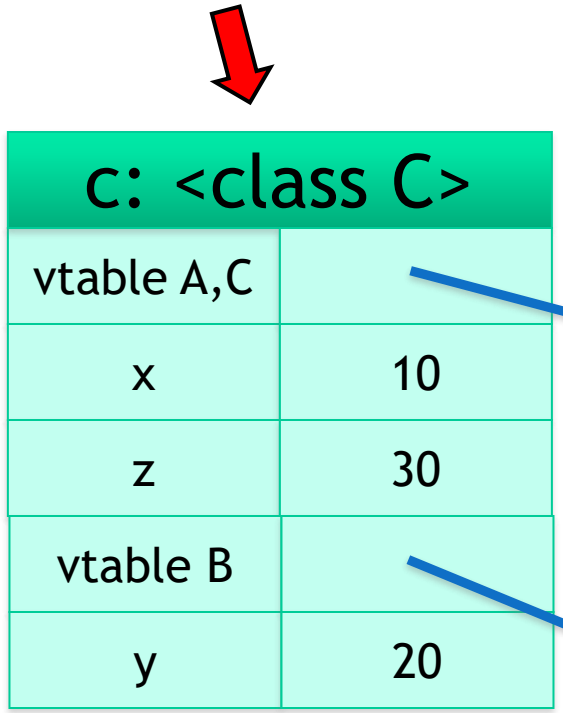
```cpp
int main() {
  C c = C();
}
```

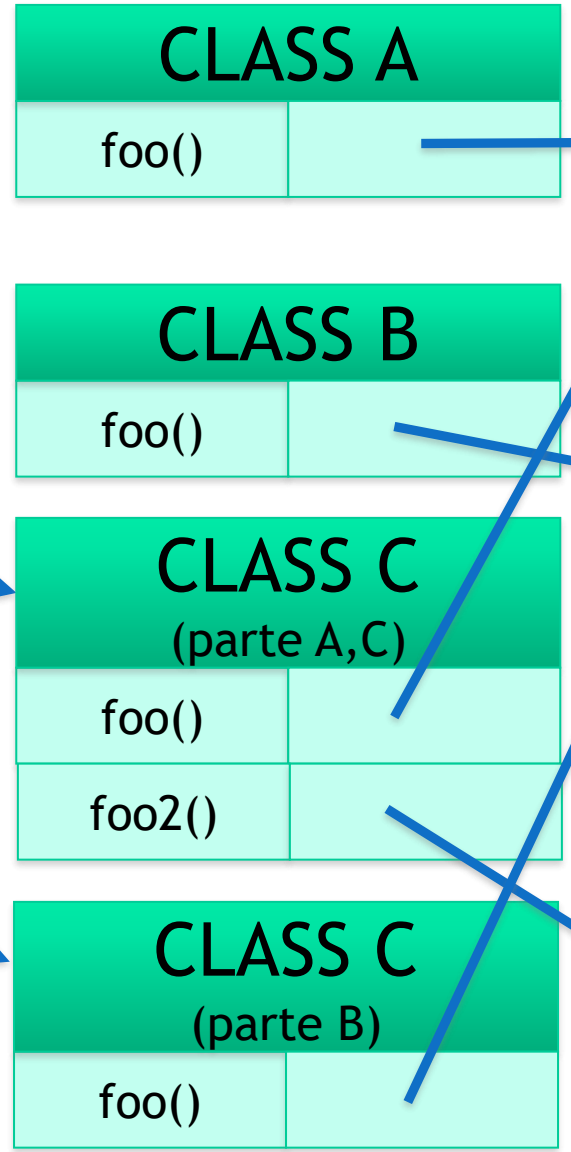



```
> g++ -o main main.cpp
> ./main
A
B
C
```

Nel runtime di C++

Descrittore dell'oggetto c

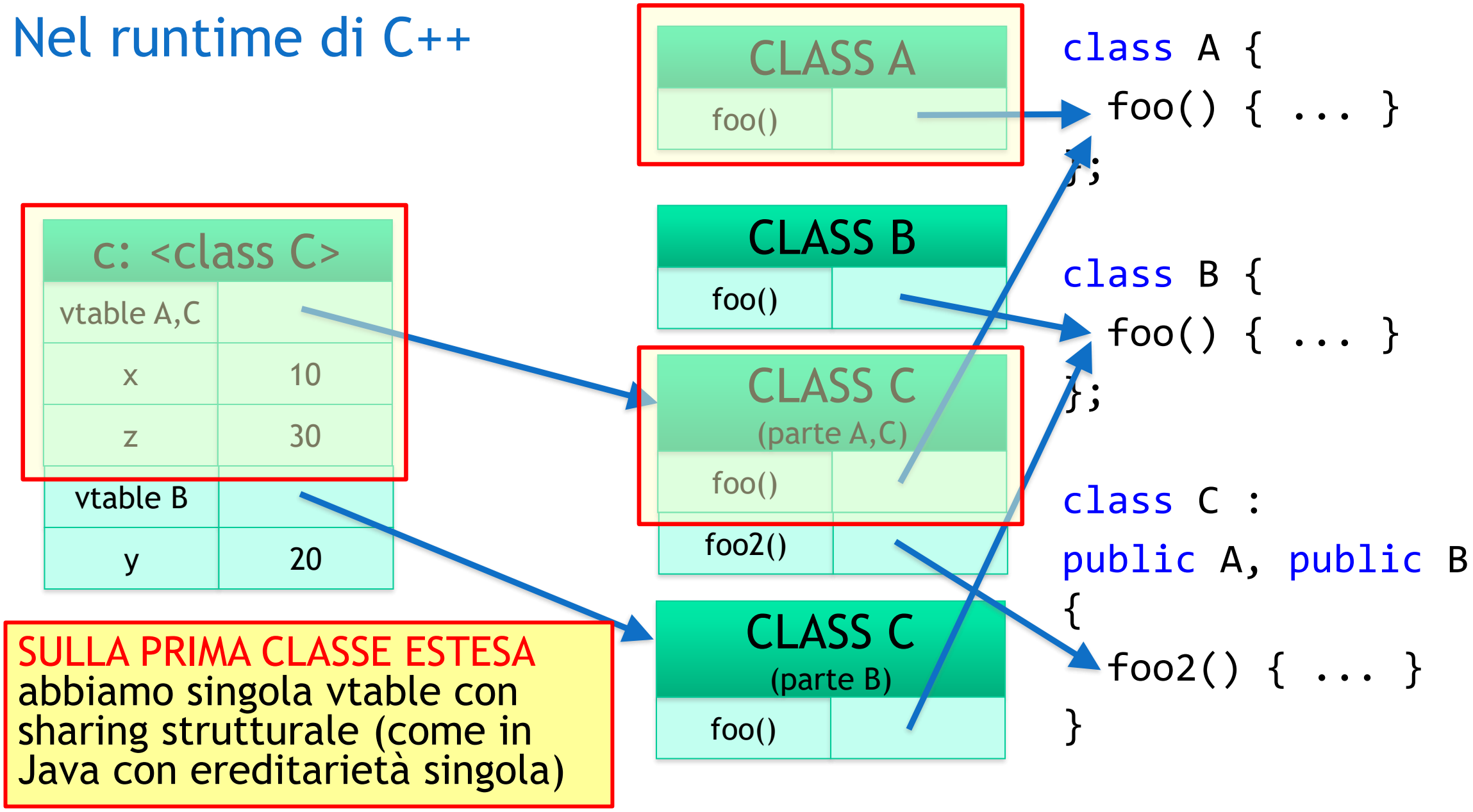




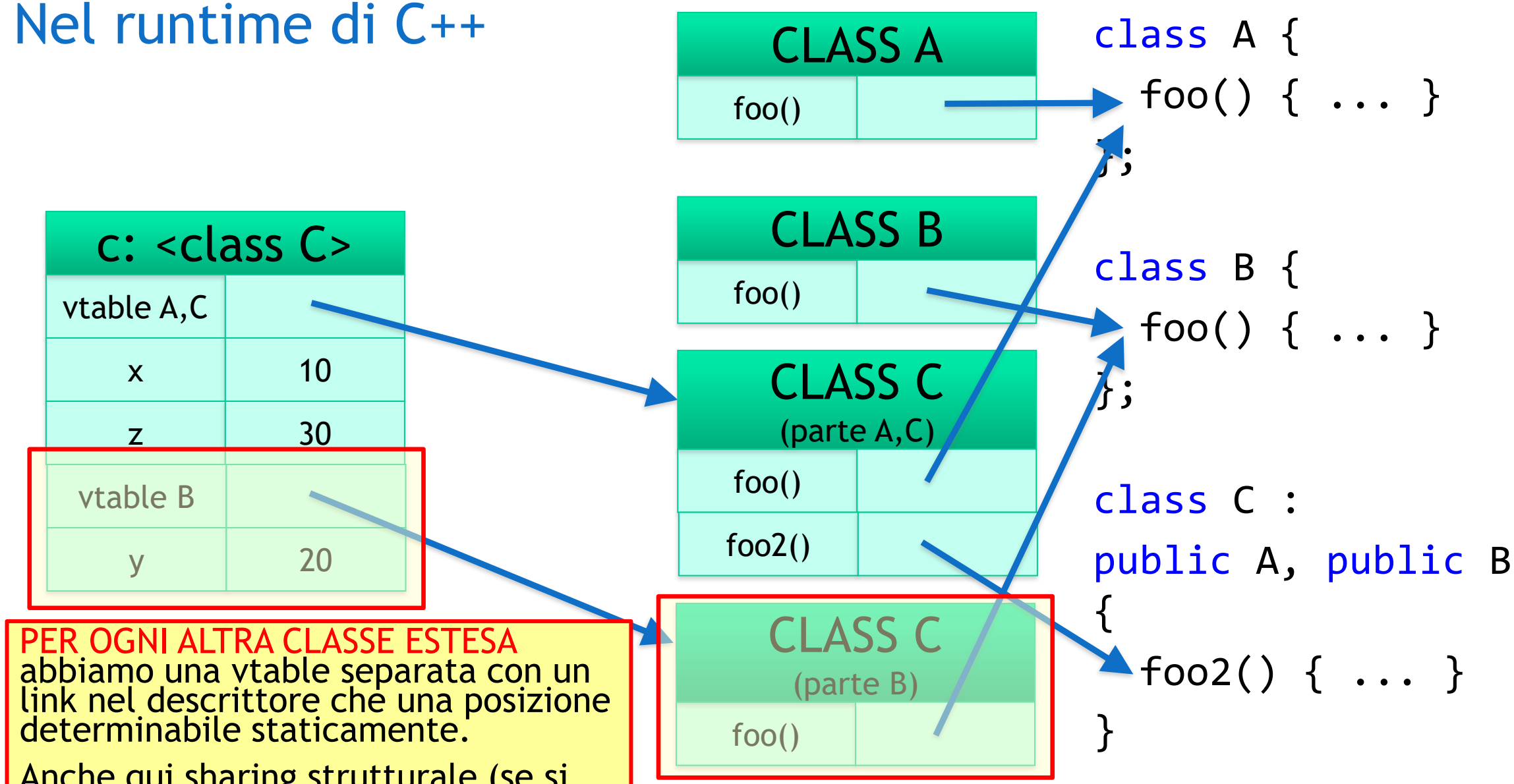

```
class A {
  foo() { ... }
};

class B {
  foo() { ... }
};

class C :
public A, public B
{
  foo2() { ... }
}
```

Tabelle dei metodi
(dispatch vectors - vtables)

Nel runtime di C++



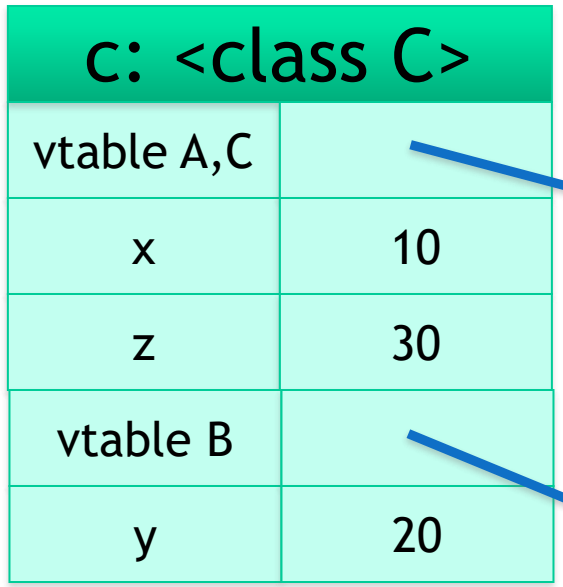
Nel runtime di C++

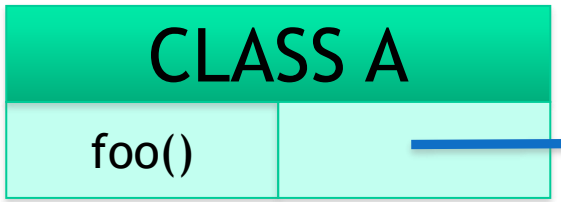


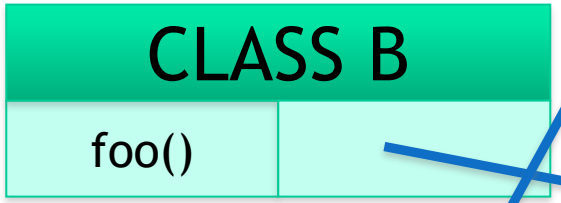
PER OGNI ALTRA CLASSE ESTESA abbiamo una vtable separata con un link nel descrittore che una posizione determinabile staticamente.

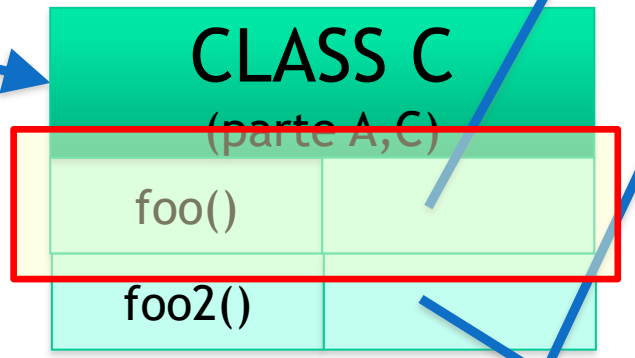
Anche qui sharing strutturale (se si usa c con tipo statico B si usa solo questa parte del descrittore)

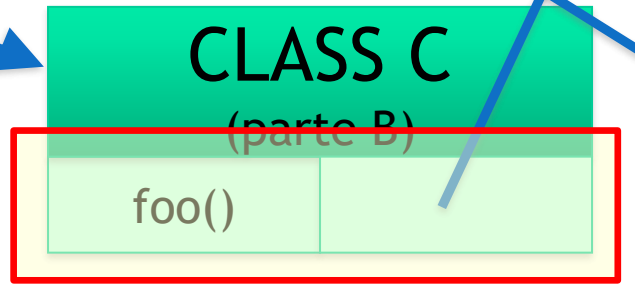
Nel runtime di C++











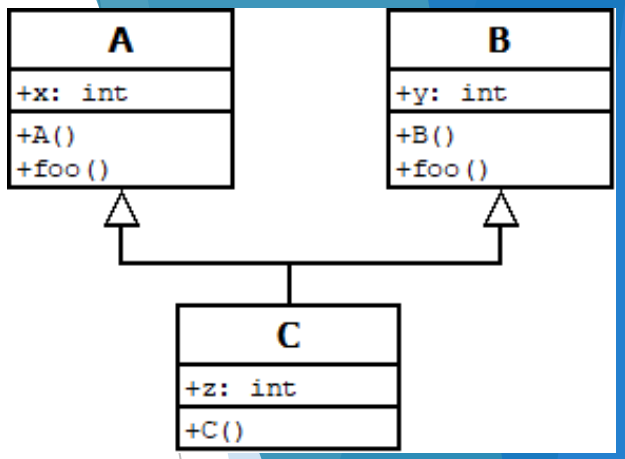

```
class A {
  foo() { ... }
};

class B {
  foo() { ... }
};

class C :
public A, public B
{
  foo2() { ... }
}
```

ENRAMBE LE IMPLEMENTAZIONI di foo() sono potenzialmente raggiungibili dal descrittore di c

C++

```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 {
  int z = 30;
  public:
    C() { cout << "C" << endl; }
    void foo2() { cout << "foo2 C" << endl; }
};
```

```cpp
int main() {
  C c = C();
  c.foo()  // CHIAMATA AL METODO foo()
}
```

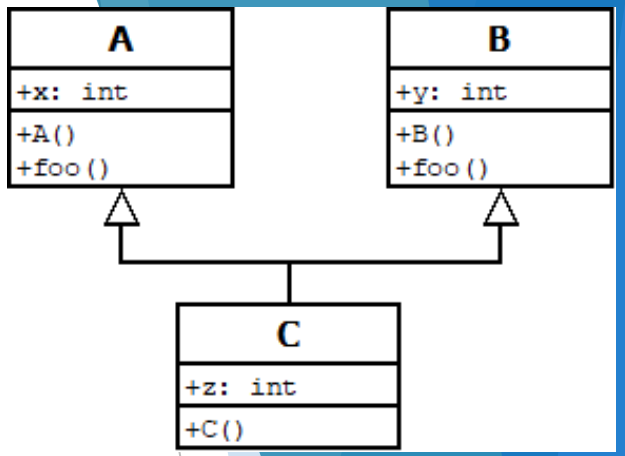



```
> g++ -o main main.cpp
main.cpp: In function ‘int main()’:
main.cpp:24:5: error: request for member ‘foo’ is ambiguous
   c.foo();
     ^~~
main.cpp:13:10: note: candidates are: void B::foo()
     void foo() { cout << "foo" << endl; }
          ^~~
main.cpp:7:10: note:                 void A::foo()
     void foo() { cout << "foo" << endl; }
          ^~~
```

C++

```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 {
  int z = 30;
  public:
    C() { cout << "C" << endl; }
    void foo2() { cout << "foo2 C" << endl; }
};
```

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

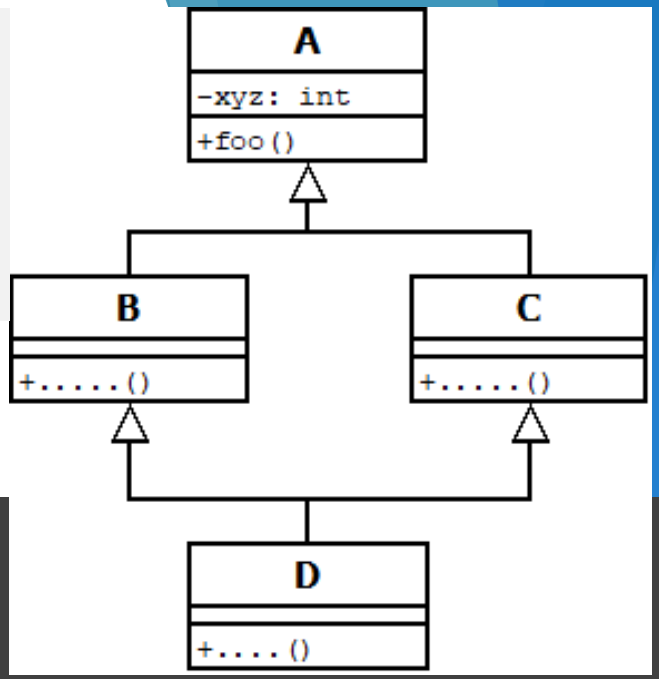



```
> g++ -o main main.cpp
> ./main
A
B
C
foo A
```

C++

```cpp
#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; }
};
```

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

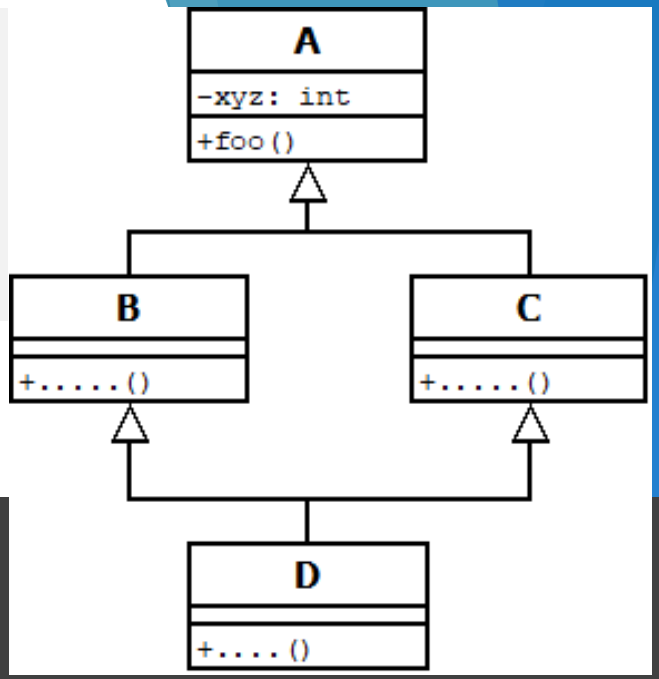



```
> g++ -o main main.cpp
> ./main
A
B
A
C
D
```

C++

```cpp
#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; }
};
```

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

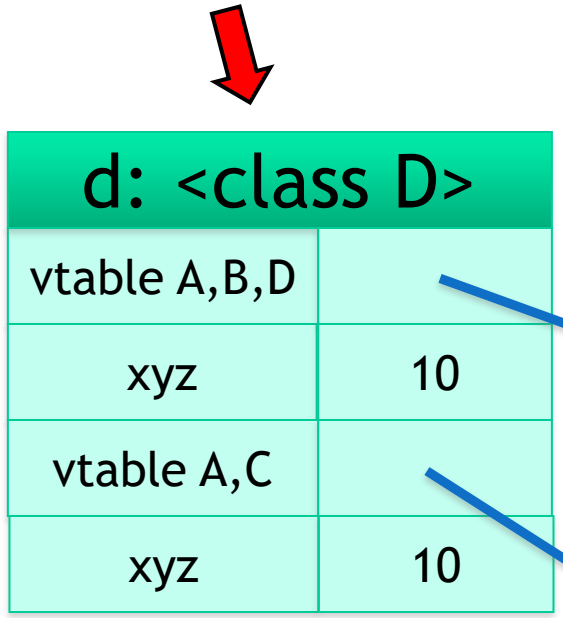


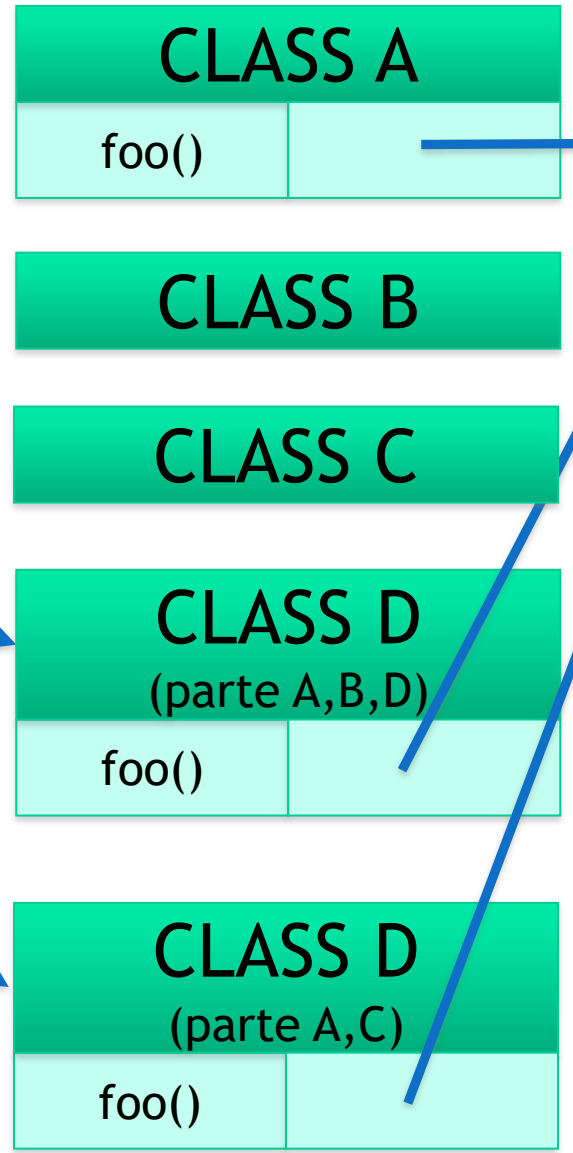

```
> g++ -o main main.cpp
> ./main
A
B
A
C
D
```

CREA DUE SOTTO-ISTANZE DI A

Nel runtime di C++

Descrittore dell'oggetto d

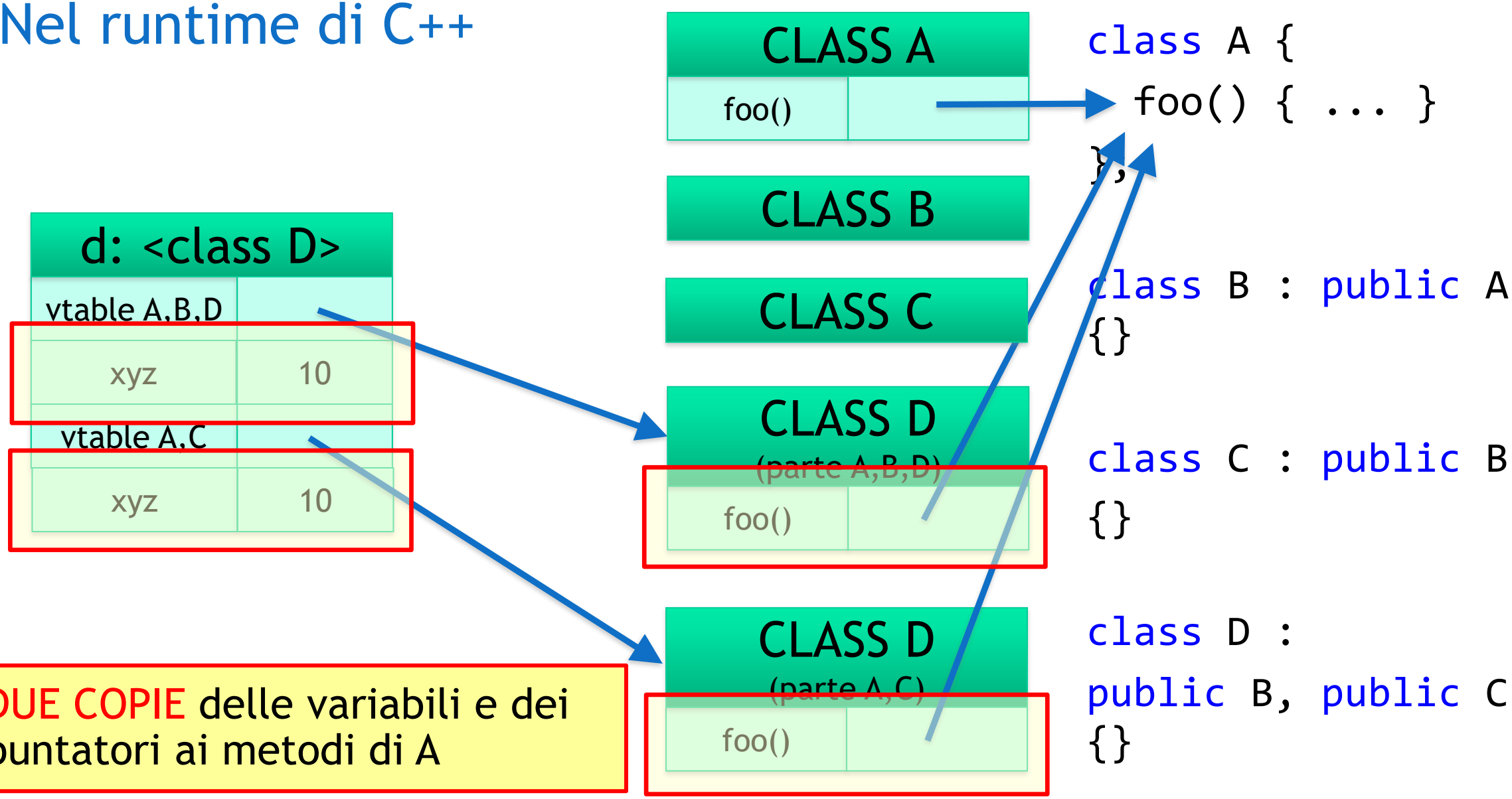





```
class A {
  foo() { ... }
};

class B : public A
{}

class C : public B
{}

class D :
public B, public C
{}
```

Tabelle dei metodi
(dispatch vectors - vtables)

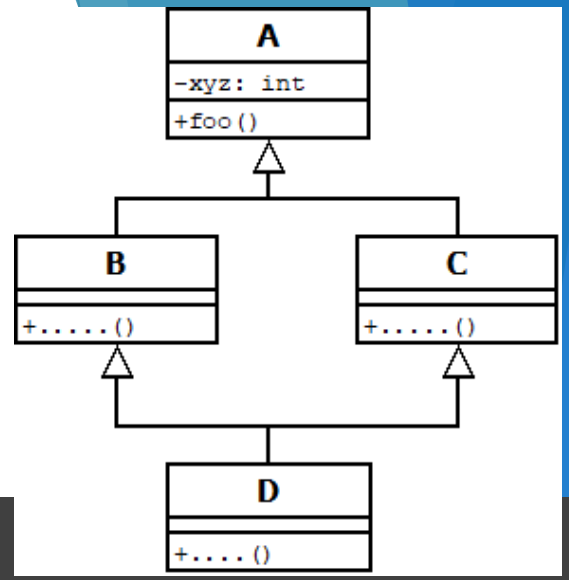
Nel runtime di C++



C++

```cpp
#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; }
};
```

```cpp
int main() {
  D d = D();
  d.foo(); // CHIAMATA AL METODO foo()
}
```

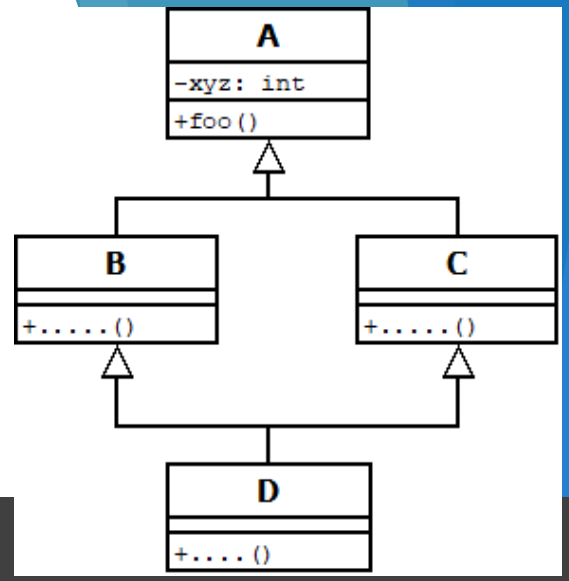



```
> g++ -o main main.cpp
main.cpp: In function ‘int main()’:
main.cpp:28:5: error: request for member ‘foo’ is ambiguous
   d.foo();
     ^~~
main.cpp:8:10: note: candidates are: void A::foo()
     void foo() { cout << "foo A" << endl; }
          ^~~
main.cpp:8:10: note:                 void A::foo()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

C++

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

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

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

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

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

```cpp
int main() {
  D d = D();
  d.foo(); // CHIAMATA AL METODO foo()
}
```

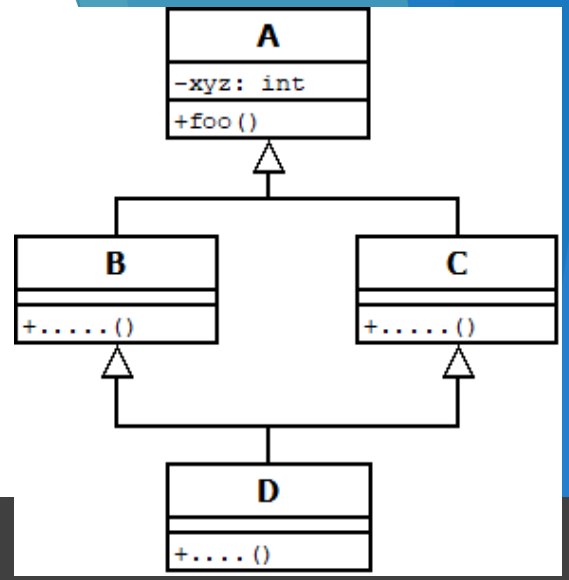



```
> g++ -o main main.cpp
> ./main
A
B
C
D
foo A
```

C++

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

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

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

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

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

```cpp
int main() {
  D d = D();
  d.foo(); // CHIAMATA AL METODO foo()
}
```



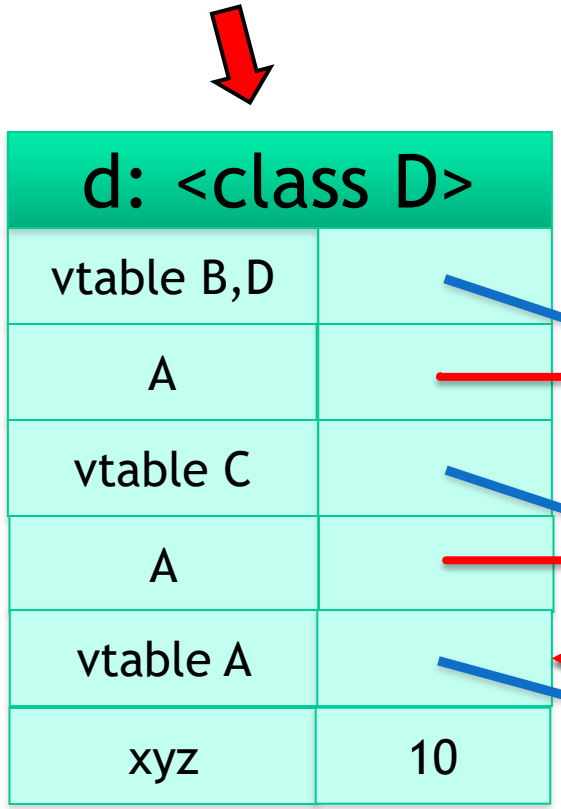

```
> g++ -o main main.cpp
> ./main
A
B
C
D
foo A
```

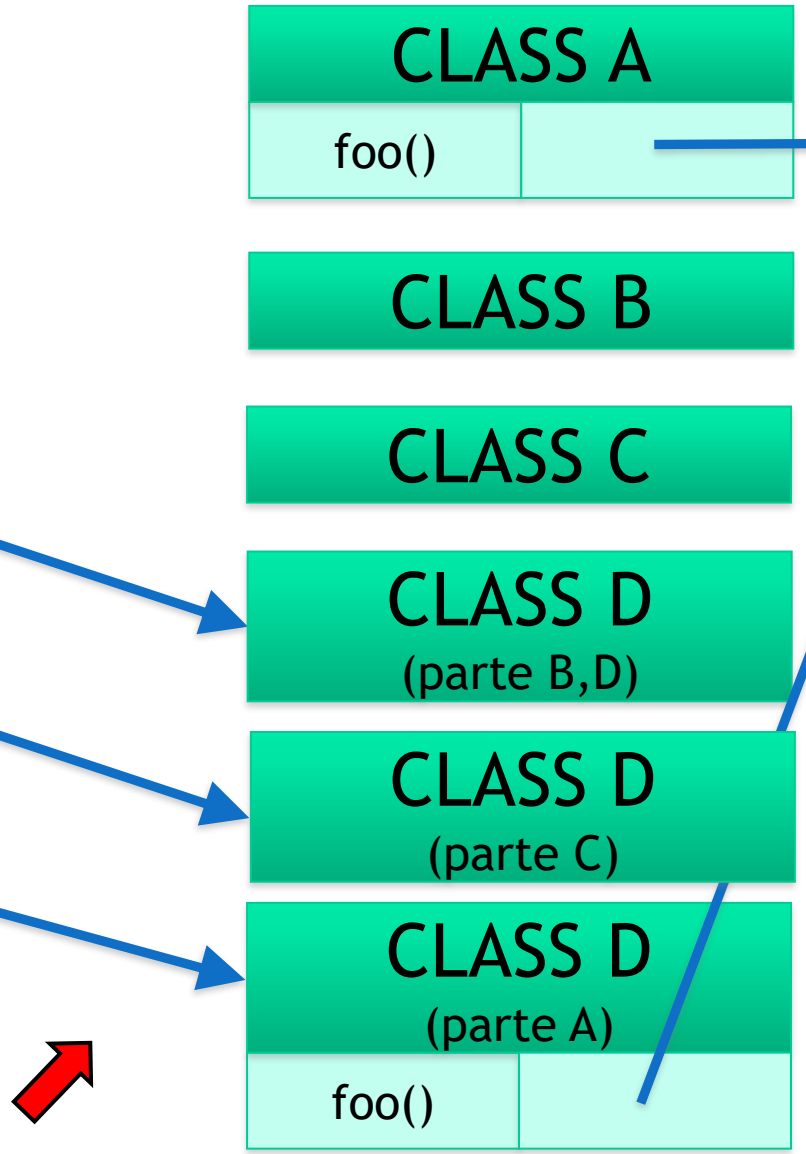
virtual inheritance:

crea un'unica copia dei "sub-objects" comuni

Nel runtime di C++

Descrittore dell'oggetto d

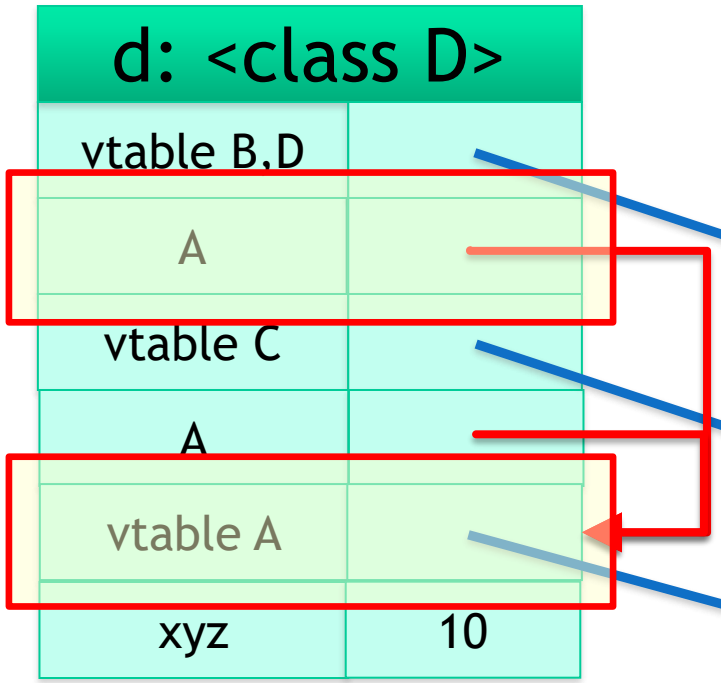


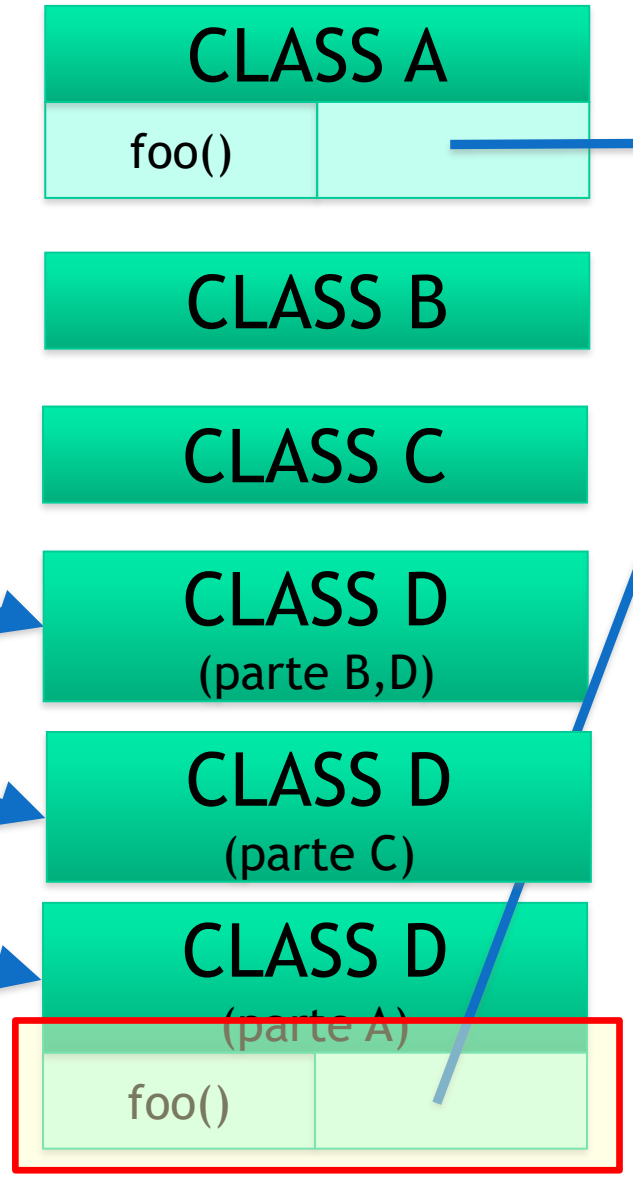



```
class A {
  foo() { ... }
};

class B : public A
{}

class C : public B
{}

class D :
public B, public C
{}
```

Tabelle dei metodi
(dispatch vectors - vtables)

Nel runtime di C++

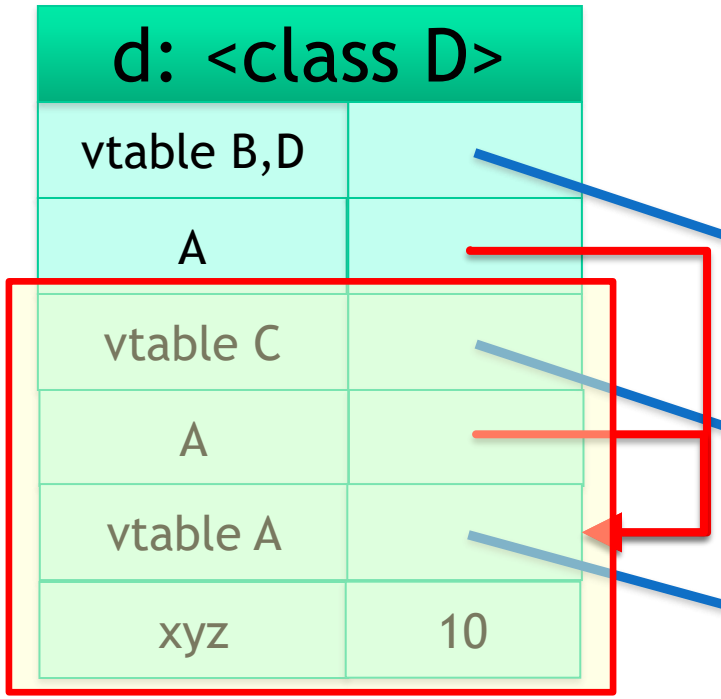





```
class A {
  foo() { ... }
};

class B : public A
{}

class C : public B
{}

class D :
public B, public C
{}
```

CHIAMARE UN METODO DI A ha un piccolo overhead (indirezione)

Nel runtime di C++




```
class A {
  foo() { ... }
};

class B : public A
{}

class C : public B
{}

class D :
public B, public C
{}
```

La seconda copia del puntatore A serve per quando si usa d con tipo statico C (con un offset sul descrittore)

Osservazioni sulla soluzione di C++

Vantaggi:

- C++ supporta l'ereditarietà multipla senza restrizioni

Svantaggi:

- Il programmatore deve essere consapevole di quello che fa

Prestazioni:

- L'ereditarietà con replicazione (non virtual) consente un accesso ai metodi in tempo costante, ma richiede di fare attenzione quando le gerarchie sono complicate (diamond problem)
- L'ereditarietà con condivisione (virtual) risolve il diamond problem, ma ha un overhead (per questo C++ lascia questa possibilità come opzione e non la usa di default)

Inheritance in Java

Come già visto in precedenza, Java prevede:

- ereditarietà singola + implementazione multipla di interfacce
- soluzione più limitata ma più semplice da tenere sotto controllo per il programmatore

Come in C++, il meccanismo dei dispatch vector e dello sharing strutturale consentono di:

- determinare la posizione del puntatore al metodo anche quando il tipo apparente (statico) è un supertipo del tipo effettivo (dinamico)
- eseguire il dispatching dei metodi in tempo costante

Inheritance in Java (interfacce)

Quando però una classe implementa più di una interfaccia lo sharing strutturale non si può usare:

```
interface Shape {                          D.V.Index
  void setCorner(int w, Point p);                0
}

interface Color {
  float get(int rgb);                            0
  void set(int rgb, float value);                1
}

class Blob implements Shape, Color {
  void setCorner(int w, Point p) {…}             0?
  float get(int rgb) {…}                         0?
  void set(int rgb, float value) {…}             1?
}
```

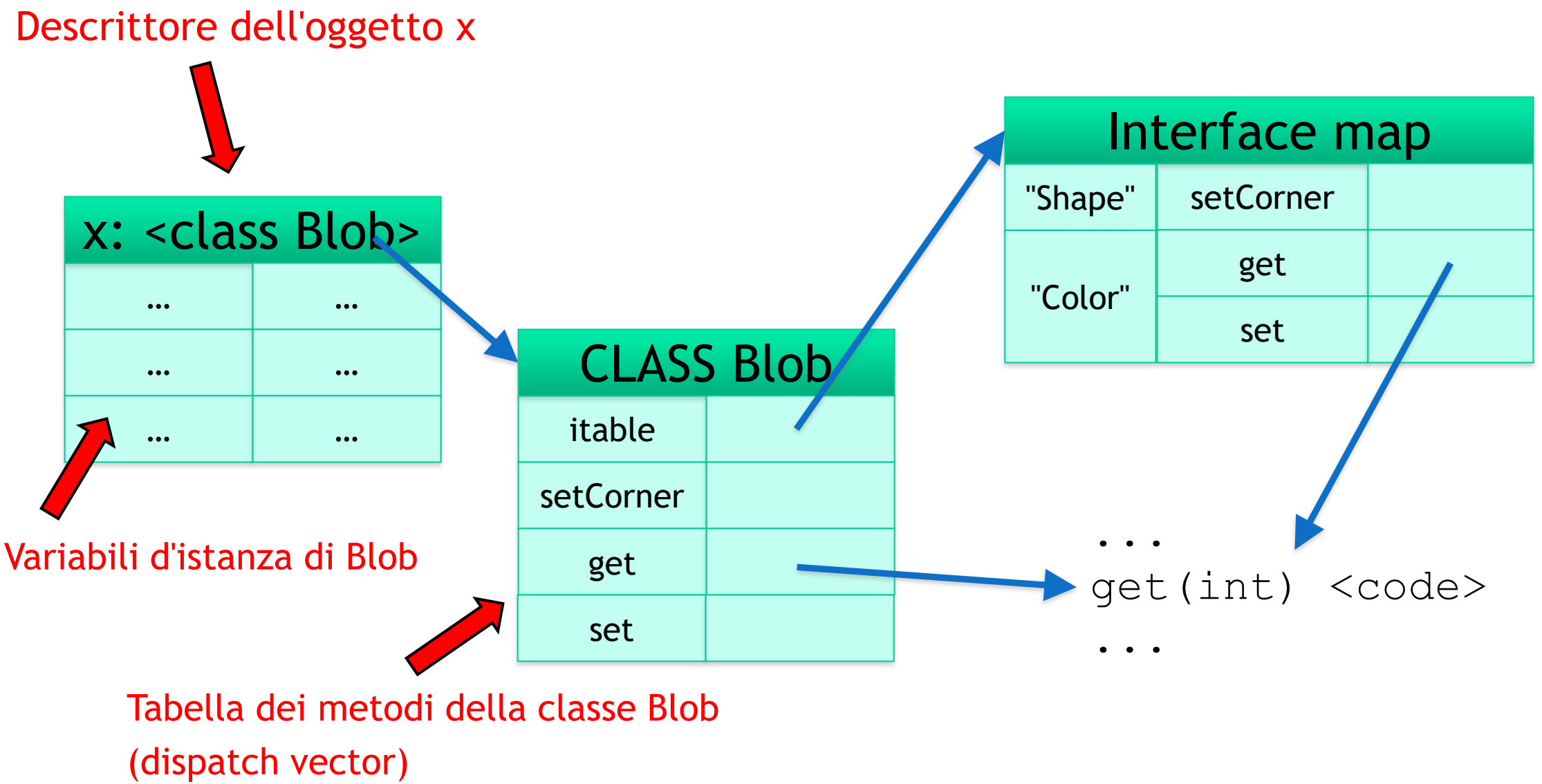
Impossibile usare gli stessi indici dei dispatch vectors di entrambe le interfacce…

Inheritance in Java (interfacce)

Per gestire le interfacce multiple, Java usa una tabella delle interfacce (itable)

- Ogni classe ha una itable con dentro i dispatch vectors di tutte le interfacce implementate
- La itable contiene una copia dei link al codice dei metodi implementati
- Se il tipo apparente dell'oggetto è quello della classe (es. Blob) la chiamata del metodo si svolge normalmente (`invokevirtual` nel bytecode)
- Se il tipo apparente dell'oggetto è quello dell'interfaccia, il compilatore compila il codice in modo che il metodo sia raggiunto tramite la itable (`invokeinterface` nel bytecode)

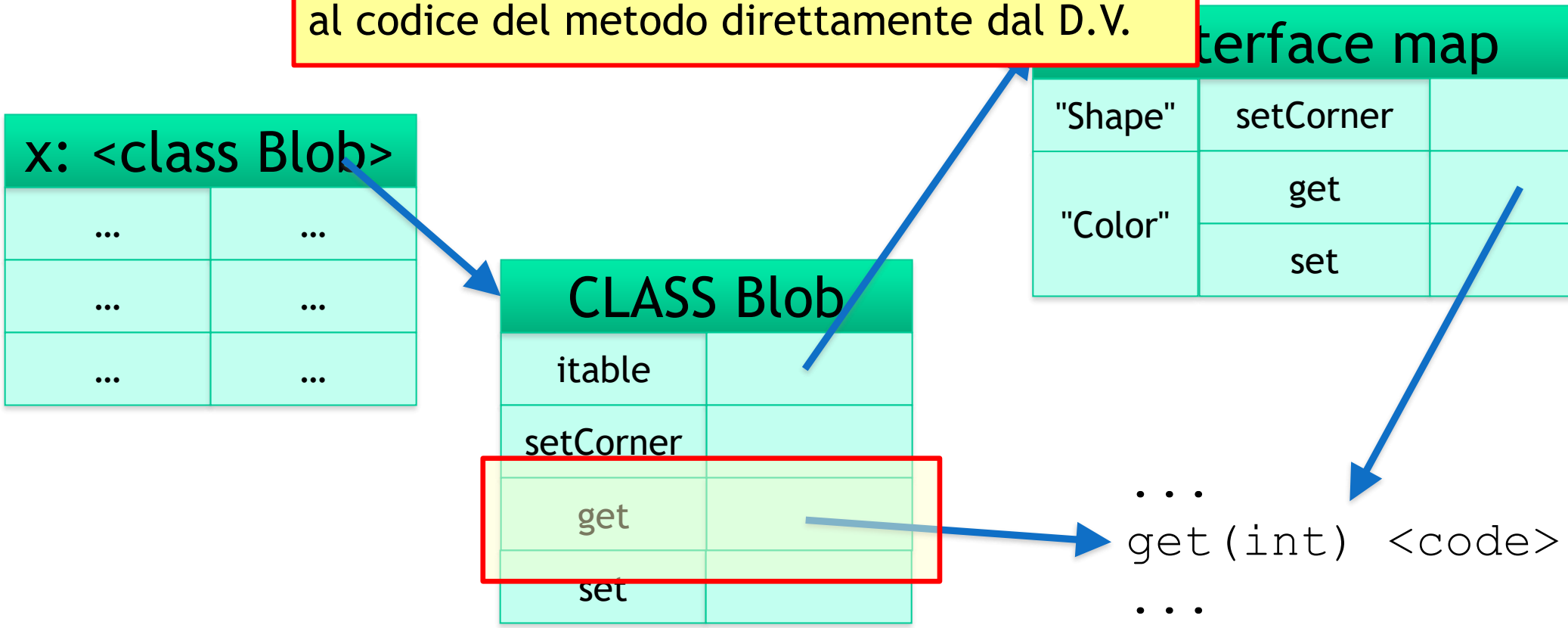
Nella JVM (a runtime)



Nella JVM (a runtime)

caso Blob x = new Blob()

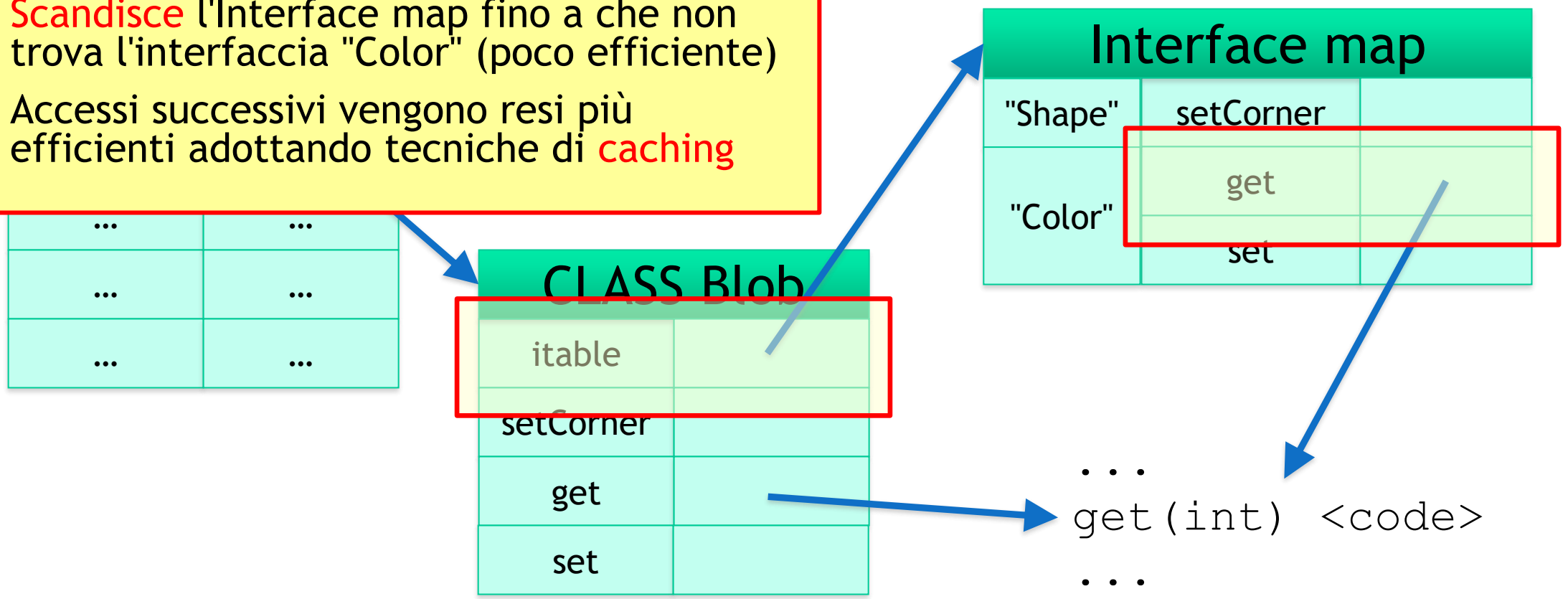
il compilatore genera il bytecode che accede al codice del metodo direttamente dal D.V.



caso Color x = new Blob()

il compilatore (che non sa che il tipo effettivo è Blob) genera il bytecode che accede al codice del metodo tramite itable

- Scandisce l'Interface map fino a che non trova l'interfaccia "Color" (poco efficiente)
- Accessi successivi vengono resi più efficienti adottando tecniche di caching



Osservazioni sulla soluzione di Java

Vantaggi:

- La soluzione adottata da Java per la gestione delle interfacce è completamente trasparente al programmatore (può non sapere…)

Svantaggi:

- Java consente un uso limitato dei meccanismi di ereditarietà

Prestazioni:

- L'accesso ai metodi tramite itable può essere poco efficiente se la classe implementa varie interfacce
- I meccanismi di caching migliorano le prestazioni dopo il primo accesso

Interfacce e default methods

Problema: In Java 8 (2014) per poter estendere le API di Java con elementi di programmazione funzionale (lambda espressioni) senza compromettere il funzionamento dei vecchi programmi (backward compatibility) è stata introdotta la possibilità di aggiungere implementazioni di default dei metodi nelle interfacce!

JAVA

```
public interface Example {
    public void a();
    default public void b() {
        System.out.println("Implementazione di default di b()");
    }
}
```

DI FATTO: ereditarietà multipla con tutti i problemi che comporta (es. diamond problem)

SOLUZIONE (BRUTALE): in caso di ambiguità il compilatore da errore e il programmatore deve modificare il codice dei metodi in conflitto

Il lavoro del compilatore…

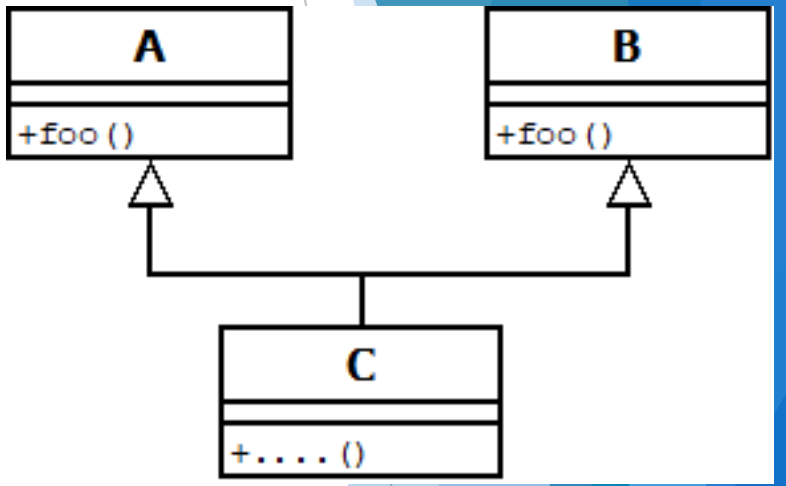
Le soluzioni di C++ e Java richiedono una fase di compilazione:

- per controllare staticamente che le chiamate dei metodi non siano ambigue (quando c'è più di una implementazione disponibile)
- per generare correttamente le strutture dati del runtime (dispatch vector/vtable e itable di ogni classe)

E per i linguaggi interpretati (senza compilazione)?

In assenza di compilazione, la gestione dell'ereditarietà nel runtime non può basarsi su tabelle dei metodi precostruite

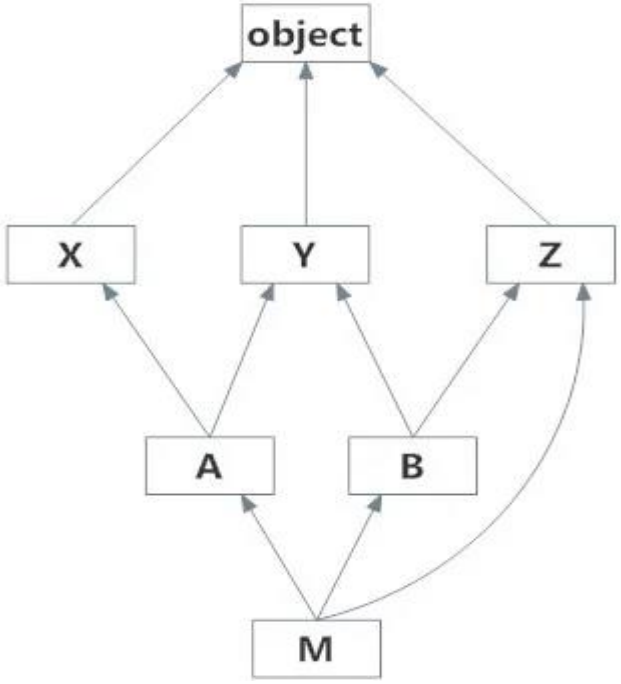
- Per trovare il codice di un metodo ereditato è necessario risalire l'albero della gerarchia di classi
- Ma: in caso di ereditarietà multipla la gerarchia è descritta da un grafo, non da un albero
  - Cammini di risalita diversi possono portare a implementazioni diverse dello stesso metodo…




```
c = new C();
c.foo(); // quale eseguire?
```

Method Resolution Order (MRO) in Python (accenno)

Il linguaggio Python usa un metodo di linearizzazione della gerarchia di classi per risolvere il problema del dispatching dei metodi con multiple inheritance e senza precompilazione



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

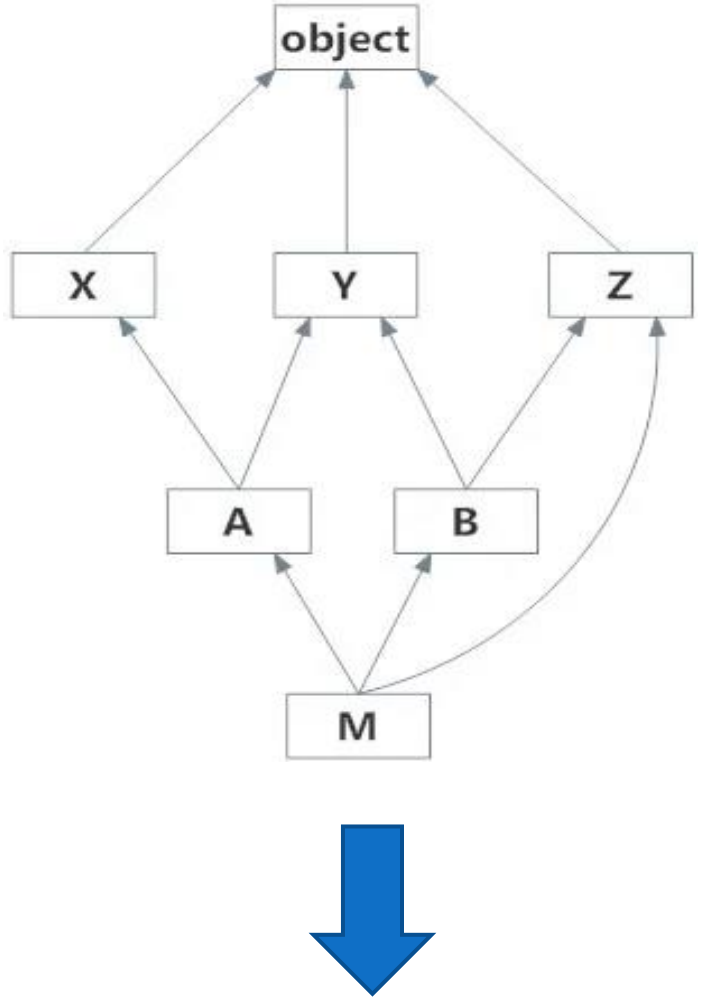
Un metodo invocato su un'istanza di M viene cercato nelle classi nell'ordine in cui sono nella lista!

Method Resolution Order (MRO) in Python (accenno)

Il calcolo del MRO si basa sull'algoritmo C3, che garantisce:

- determinismo (unico ordinamento)
- conservazione dell'ordinamento locale (se una classe estende c1,c2, …, cn, l'ordinamento calcolato dovrà prevedere queste classi in quest'ordine)
- monotonia (se c1 è sottoclasse di c2, nell'ordinamento calcolato c1 dovrà venire prima di c2)

Non entriamo nel merito dell'algoritmo…



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

PYTHON

```python
class X:
  pass  # non fa nulla...

class Y:
  pass

class Z:
  pass

class A(X, Y):  # A estende X e Y
  pass

class B(Y, Z):
  pass

class M(A, B, Z):
  pass

print(M.mro());  # stampa l'ordine
```

```
> python main.py

[<class '__main__.M'>, <class '__main__.A'>, <class
'__main__.X'>, <class '__main__.B'>, <class '__main__.Y'>,
<class '__main__.Z'>, <class 'object'>]
```

Method Resolution Order (MRO) in Python (accenno)

Osservazioni:

- Esistono gerarchie di classi non linearizzabili (errore a runtime)
- Il programmatore dovrà essere consapevole di quale versione di un metodo chiamato verrà eseguita in base all'ordinamento
  - In fase di sviluppo è utile impostare le classi è chiamare mro() per sapere quale sarà l'ordine di linearizzazione

PYTHON

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

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

Inheritance VS Composition: Mixin

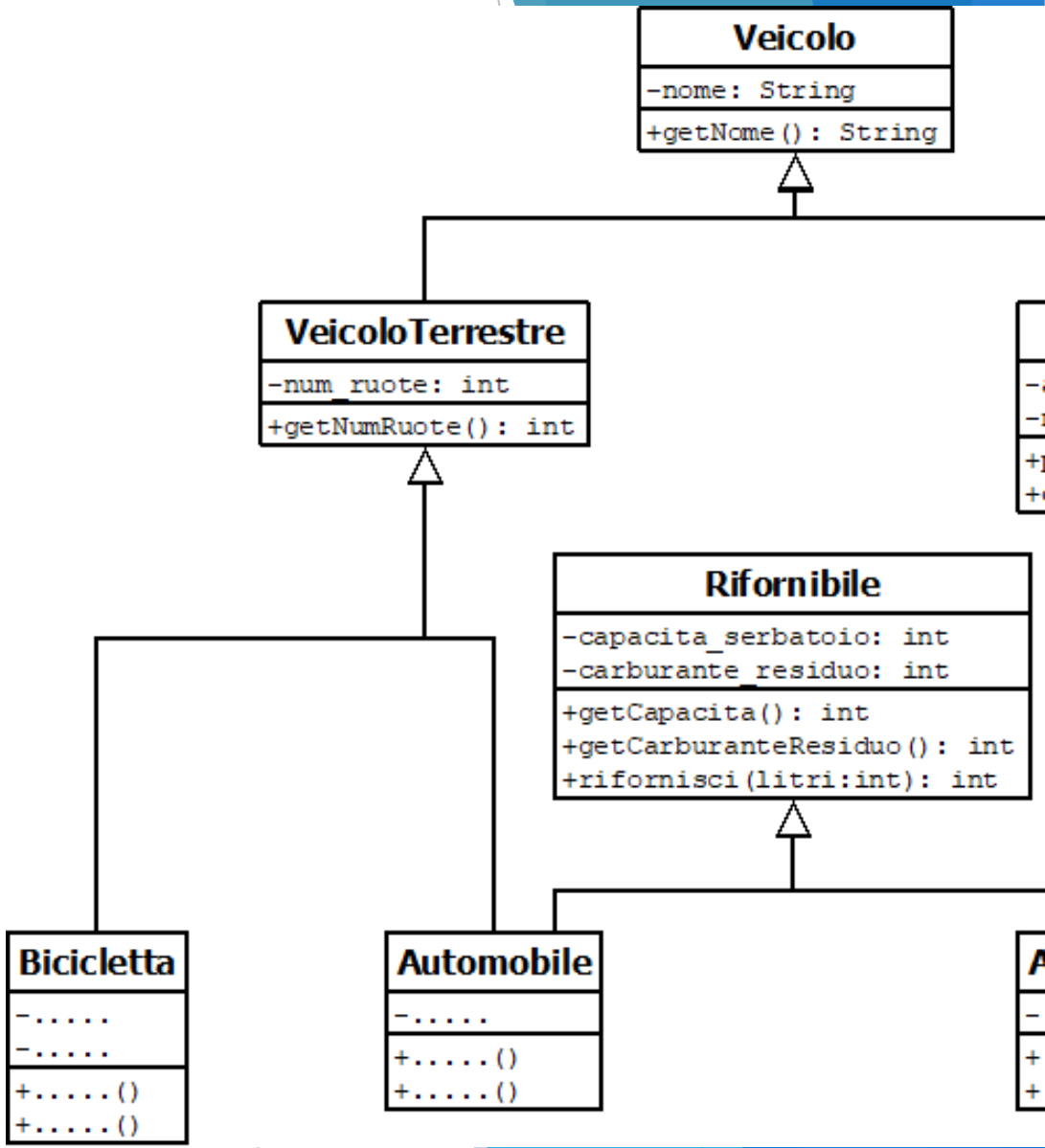
Un approccio concettualmente diverso al problema dell'ereditarietà multipla è quello rappresentato dai mixin.

- Idea: anziché ereditare da altre (super)classi, una classe potrebbe essere risultato della composizione di altre classi
- Un mixin è un componente che può essere mescolato (mixed-in) ad una classe esistente
- In alcuni linguaggi (es. Scala) ci sono i traits: concetto simile

Mixin

Ad esempio:

- **`Rifornibile`** potrebbe essere definita come mixin invece che come classe
  - insieme di funzionalità utilizzabili in altre classi
  - non se ne creano oggetti
- **`Automobile`** potrebbe essere vista come
  - sottoclasse di **`VeicoloTerrestre`**
  - composta con il mixin **`Rifornibile`**
- Ereditarietà singola e con possibilità di aggiungere (anche facendo overriding) i metodi del mixin a quelli della superclasse



Mixin

Molti linguaggi che prevedono ereditarietà multipla possono adottare (o simulare) i mixin come stile di programmazione:

- Ad esempio: in Python i mixin sono convenzionalmente realizzati (dal programmatore) come classi che non hanno superclassi (no diamond problem) e che contengono un insieme di metodi concettualmente utilizzabili in classi diverse

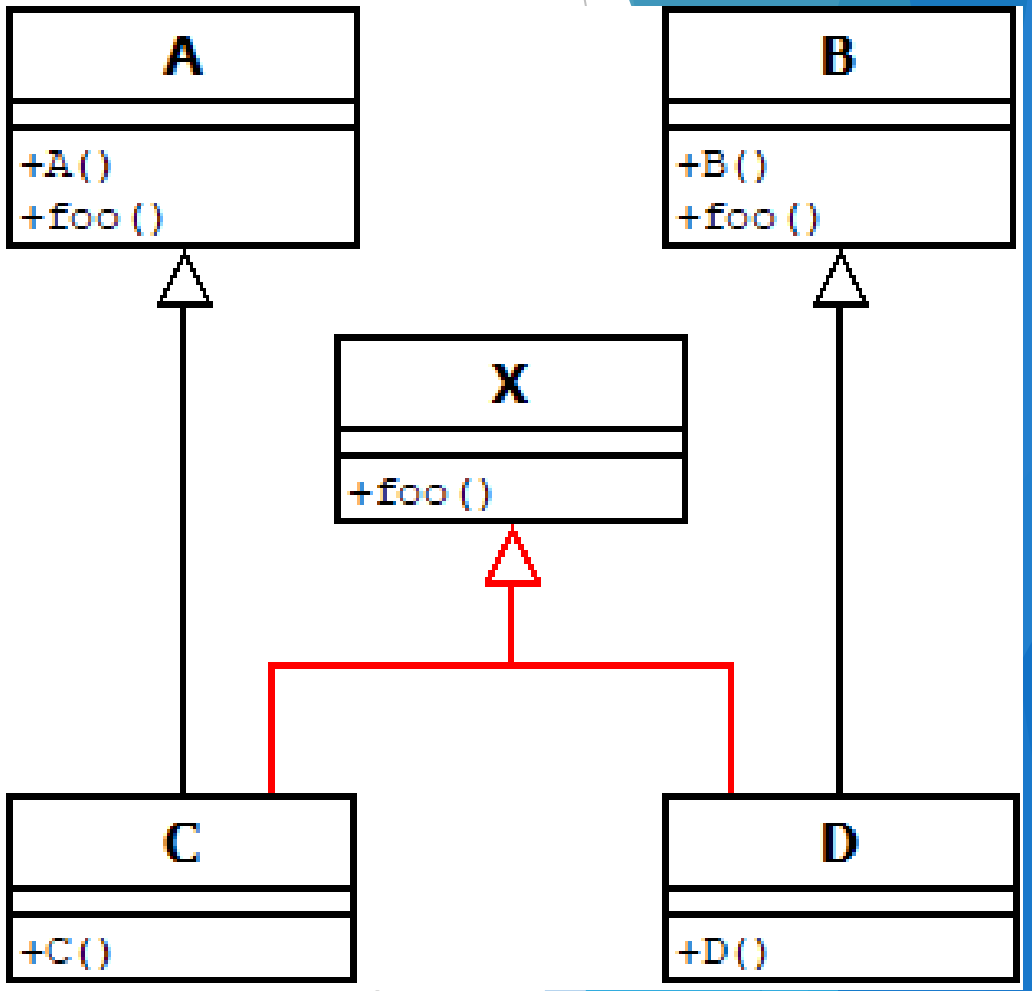
Alcuni linguaggi iniziano ad introdurre costrutti linguistici per la realizzazione di mixin

- Esempio: Dart (linguaggio sviluppato dal 2011 da Google come possibile alternativa a JavaScript in ambito web)

Esempio di Mixin in Dart

Proviamo ad impostare questa gerarchia di classi in Dart

- X è un mixin



DART

```dart
class A {
  A() { print('constructor A'); }
  void foo() { print('foo A'); }
}

class B {
  B() { print('constructor B'); }
  void foo() { print('foo B'); }
}

mixin X {  // mixin X
  void foo() { print('foo X'); }
}


// estende A e compone con X
class C extends A with X {
  C() { print('constructor C'); }
}


// estende B e compone con X
class D extends B with X {
  D() { print('constructor D'); }
}
```

```dart
void main() {
  C c = C();
  c.foo();

  D d = D();
  d.foo();
}
```

```
> dart main.dart
constructor A
constructor C
foo X
constructor B
constructor D
foo X
```

COME FUNZIONA:

Si può estendere una sola classe (single inheritance) e poi comporre con quanti mixin si vuole, aggiornando la definizione della classe (anche facendo overriding)

Riassumendo…

L'ereditarietà multipla è una funzionalità utile, ma complicata da realizzare

- La soluzione adottata in C++ è ottimale, ma richiede che il programmatore sia ben consapevole (per capire quando usare virtual)
- La soluzione di Java (prima di Java 8, con interfacce "tradizionali") è più semplice, ma non è vera ereditarietà multipla
  - Le interfacce con implementazioni di default dei metodi (Java 8) sono frutto di un compromesso nello sviluppo del linguaggio (per consentire l'introduzione di elementi di programmazione funzionale) ed è meglio usarle con cautela…
- Approcci di linearizzazione della gerarchia (es. Algoritmo C3 di Python) sopperiscono all'assenza di una fase di compilazione, ma rendono la chiamata di metodi poco efficiente (bisogna visitare la gerarchia linearizzata)
- Gli approcci basati su composizione (mixin) rappresentano una valida alternativa che potrebbe trovare diffusione nel prossimo futuro