Classi e oggetti: implementazione

Dai sottoprogrammi…

- Un sottoprogramma (***astrazione procedurale***)
  - meccanismo linguistico che richiede di gestire dinamicamente **ambiente** e **memoria**
- La chiamata di sottoprogramma provoca la creazione di un ambiente e di una memoria locale (record di attivazione), che esistono finché l'attivazione non restituisce il controllo al chiamante
- Ambiente locale
  - ✓ ambiente e memoria sono creati con la definizione della procedura
  - ✓ esistono solo per le diverse attivazioni di quella procedura

… alle classi

- L'aspetto essenziale dei linguaggi a oggetti consiste nella definizione di un meccanismo che permetta di creare ambiente e memoria al momento della "attivazione" di un oggetto (la creazione dell'oggetto)
  - nel quale gli ambienti e la memoria siano persistenti (sopravvivano alla attivazione)
  - una volta creati, siano accessibili e utilizzabili da chiunque possieda il loro meccanismo di accesso ("handle")

Classi e loro istanziazione

- Tale meccanismo di astrazione linguistica è denominato ***classe***
- L'istanziazione (attivazione) della classe avviene attraverso la chiamata del costruttore, ad esempio

  `new(classe, parametri_attuali)` oppure

  `new classe(parametri_attuali)`

  – che può occorrere in una qualunque espressione

  – con la quale si passano alla classe gli eventuali parametri attuali

  – che provoca la restituzione di un ***oggetto***

Classi e istanziazione

- L'ambiente e la memoria locali dell'oggetto sono creati dalla valutazione delle **dichiarazioni**
  - le dichiarazioni di costanti e di variabili definiscono i **campi** dell'oggetto
    - se ci sono variabili, l'oggetto ha una memoria e quindi uno stato modificabile
  - le dichiarazioni di funzioni e procedure definiscono i **metodi** dell'oggetto
    - che vedono (e possono modificare) i campi dell'oggetto, per la normale semantica dei blocchi
- L'esecuzione della lista di **comandi** è l'inizializzazione dell'oggetto

Oggetti

- L'oggetto è la struttura (handle) che permette di accedere l'ambiente e la memoria locali creati permanentemente
  - attraverso l'accesso ai suoi metodi e campi
  - con l'operazione

    `Field(obj, id)` (sintassi astratta)

    `obj.id` (sintassi concreta)

- Nell'ambiente locale di ogni oggetto il nome speciale **`this`** denota l'oggetto medesimo

Oggetti e creazione dinamica di strutture dati

- La creazione di oggetti assomiglia molto (anche nella notazione sintattica) alla creazione dinamica di strutture dati tramite primitive linguistiche del tipo

  `new(type_data)`

  che provoca l'allocazione dinamica di un valore di tipo **`type_data`** e la restituzione di un puntatore a tale struttura
- Esempi: record in Pascal, struct in C

Strutture dati dinamiche

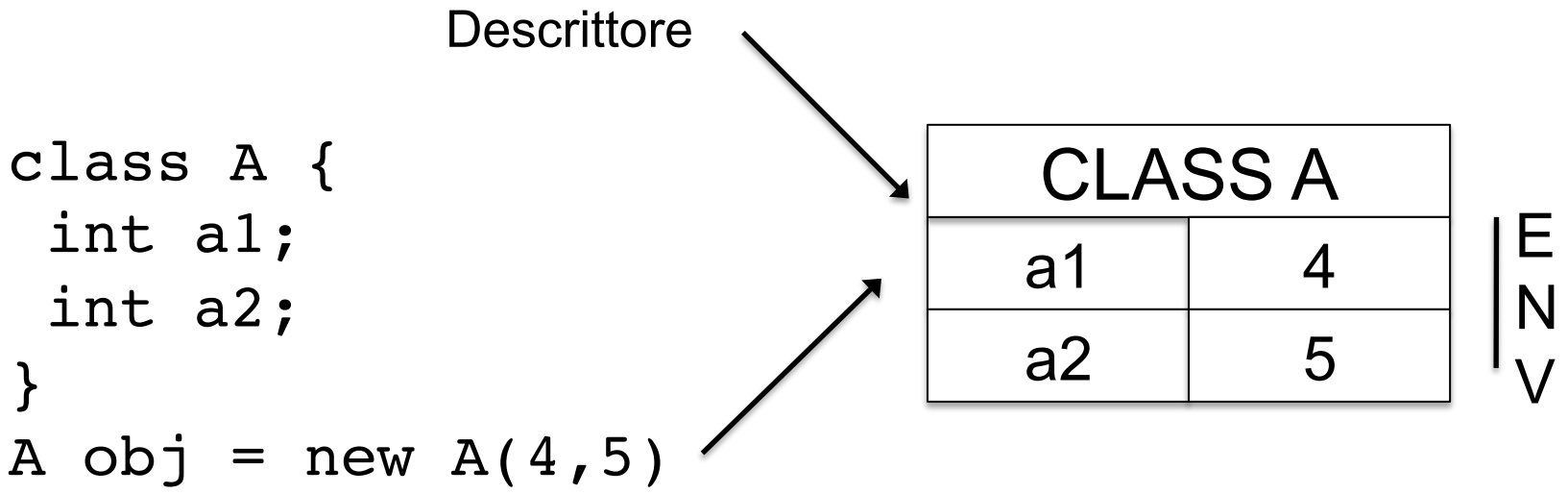
- Tale meccanismo prevede l'esistenza di una memoria a **heap**
- Strutture dati dinamiche: un caso particolare di oggetti, ma...
  - hanno una semantica *ad hoc* non riconducibile a quella dei blocchi e delle procedure
  - non consentono la definizione di metodi
  - a volte la rappresentazione non è realizzata con campi separati
  - a volte non sono davvero permanenti
    - ✓ può esistere una (pericolosissima) operazione che permette di distruggere la struttura (**`free`**)

Ingredienti del paradigma OO

- Oggetti
  - meccanismo per incapsulare dati e operazioni
- Ereditarietà
  - riuso del codice
- Polimorfismo
  - principio di sostituzione
- Dynamic binding
  - legame dinamico tra il nome di un metodo e il codice effettivo che deve essere eseguito

Implementazione: var. di istanza

- Soluzione: ambiente locale statico che contiene i binding delle variabili di istanza
  - con associato il descrittore di tipo



E la ereditarietà?

```
class A {
 int a1;
 int a2;
}

class B extends A {
 int a3
}
```

CLASS B	
a1	
a2	
a3	

Soluzione: i campi ereditati dall'oggetto vengono inseriti all'inizio nell'ambiente locale

Oggetti: ambiente locale statico

- L'utilizzo di un ambiente locale statico permette di implementare facilmente la persistenza dei valori
  - la gestione della ereditarietà (singola) è immediata
  - la gestione dello *shadowing* (variabili di istanza con lo stesso nome usata nella sottoclasse) è immediata
- Se il linguaggio prevede meccanismi di controllo statico si può facilmente implementare un accesso diretto: **indirizzo di base + offset**

Implementazioni multiple


```
interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
```

```
class IntSet1 implements IntSet {
  private List<Integer> rep;
  public IntSet1() {
    rep = new LinkedList<Integer>();}

  public IntSet1 insert(int i) {
    rep.add(new Integer(i));
    return this;}

  public boolean has(int i) {
    return rep.contains(new Integer(i));}

  public int size() {return rep.size();}
}
```

```
class IntSet2 implements IntSet {
  private Tree rep;
  private int size;
  public IntSet2() {
    rep = new Leaf(); size = 0;}

  public IntSet2 insert(int i) {
    Tree nrep = rep.insert(i);
    if (nrep != rep) {
      rep = nrep; size += 1;
    }
    return this;}

  public boolean has(int i) {
    return rep.find(i);}

  public int size() {return size;}
}
```

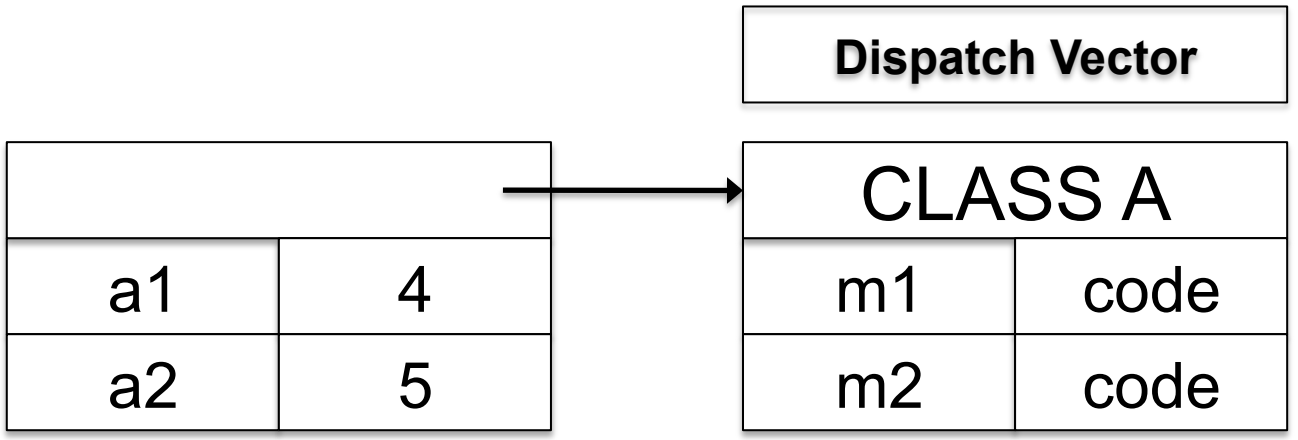
La nozione di dispatching

- Consideriamo un cliente di IntSet:
  - IntSet set = ...;
  - int x = set.size();
- Quale dei due metodi viene invocato?
  - IntSet1.size ?
  - IntSet2.size ?
- Il cliente non ha informazioni sufficienti per risolvere la questione
- Ogetti devono avere un meccanismo che permette di identificare chiaramente il codice del metodo da invocare
- Morale: invocazione di un metodo deve “passare” dagli oggetti.

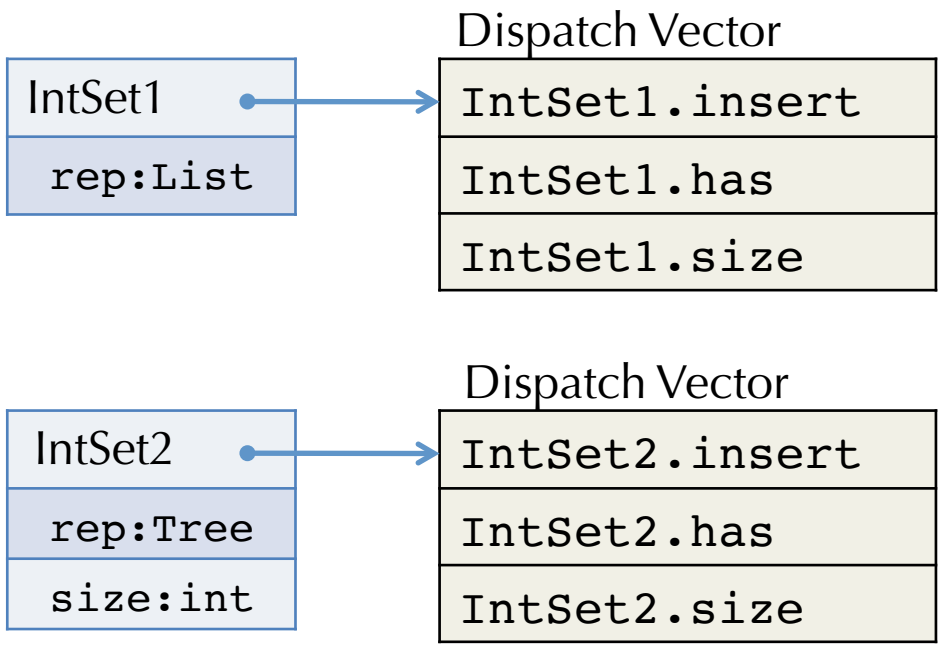
Tabella dei Metodi

- Soluzione: associare un puntatore alla tabella (*tabella dei metodi*, *vtable*, *dispatch table*) che contiene il binding dei metodi e il descrittore con altre informazioni associate alla classe

```
class A {
 int a1;
 int a2;
 int m1 …;
 void m2 …;
}
A obj = new A(4,5)
```



IntSet


Dispatch Vector
IntSet1
rep:List
IntSet1.insert
IntSet1.has
IntSet1.size
Dispatch Vector
IntSet2
rep:Tree
size:int
IntSet2.insert
IntSet2.has
IntSet2.size

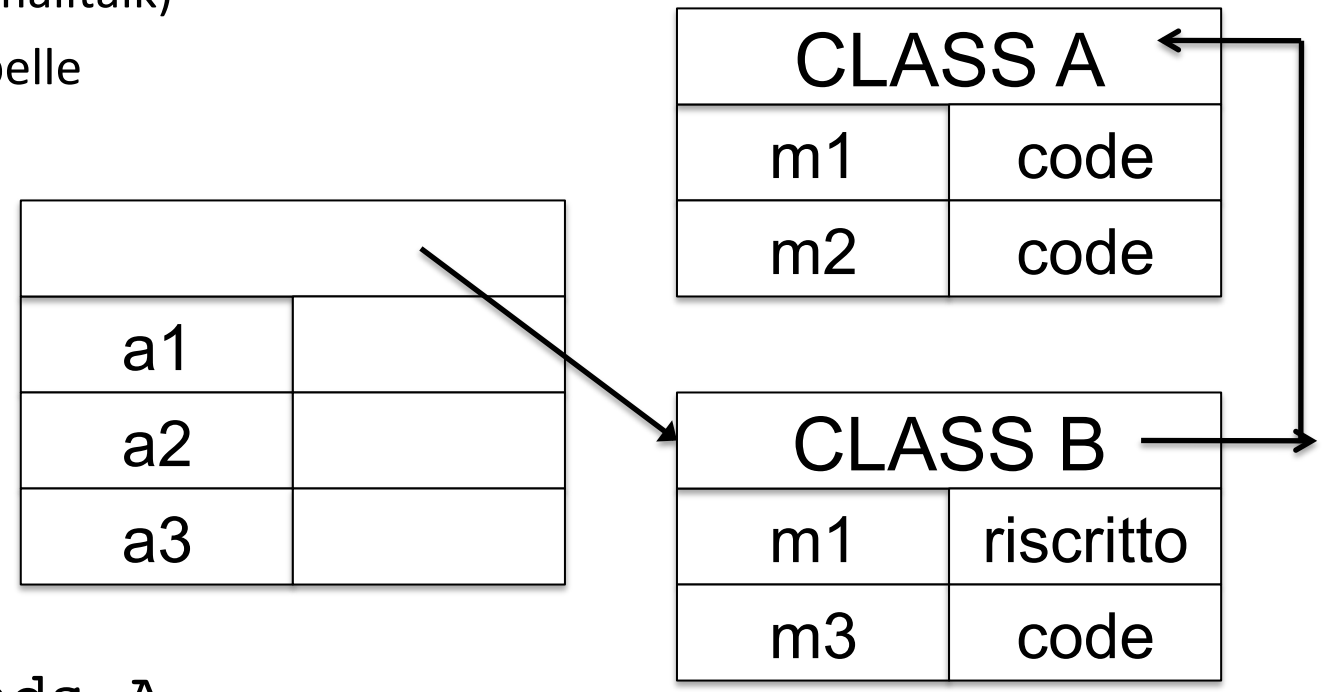
Implementazione dei metodi

- Un metodo è eseguito come una funzione (implementazione standard: AR sullo stack con variabili locali, parametri, ecc.)
- **Importante**: il metodo deve poter accedere alle variabili di istanza dell'oggetto sul quale è invocato (che non è noto al momento della compilazione)
- **L'oggetto è un parametro implicito**: quando un metodo è invocato, gli viene passato anche un puntatore all'oggetto sul quale viene invocato; durante l'esecuzione del metodo il puntatore è il **this** del metodo

Ereditarietà

- Soluzione1 (Smalltalk)
  - lista di tabelle

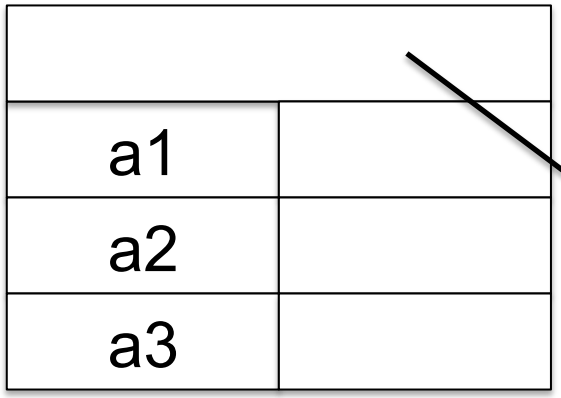
```
class A {
 int a1, a2;
 void m1 …;
 void m2 …;
}

class B extends A
 int a3;
 void m1 …;
 void m3 …;
}
```



Ereditarietà

- Soluzione 2 (C++ e Java)
  - sharing strutturale

```
class A {
 int a1, a2;
 void m1 …;
 void m2 …;
}

class B extends A
 int a3;
 void m1 …;
 void m3 …;
}
```



CLASS A	
m1	ptr_cod
m2	ptr_cod

CLASS B	
m1	*riscritto*
m2	ptr_cod
m3	ptr_cod

```
new B(…)
```

Analisi

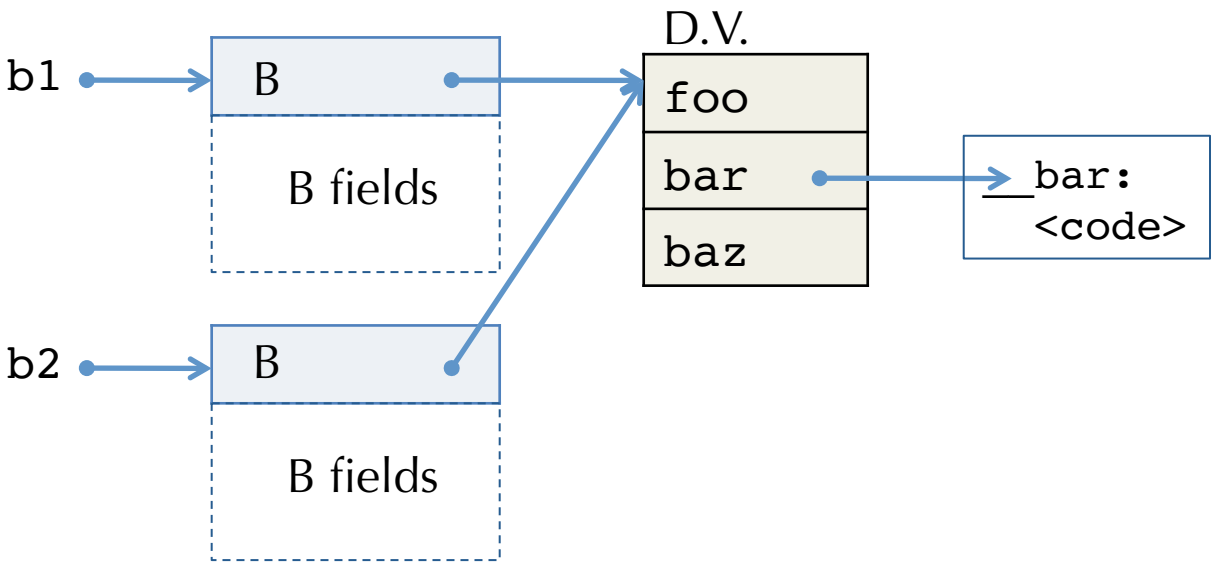
- Liste di tabelle dei metodi (Smalltalk): l'operazione di dispatching dei metodi viene risolta con una visita alla lista (overhead a run time)
- Sharing strutturale (C++): l'operazione di dispatching dei metodi si risolve staticamente andando a determinare gli offset nelle tabelle (**vtable** in C++ [virtual function table])

Discussione: Smalltalk

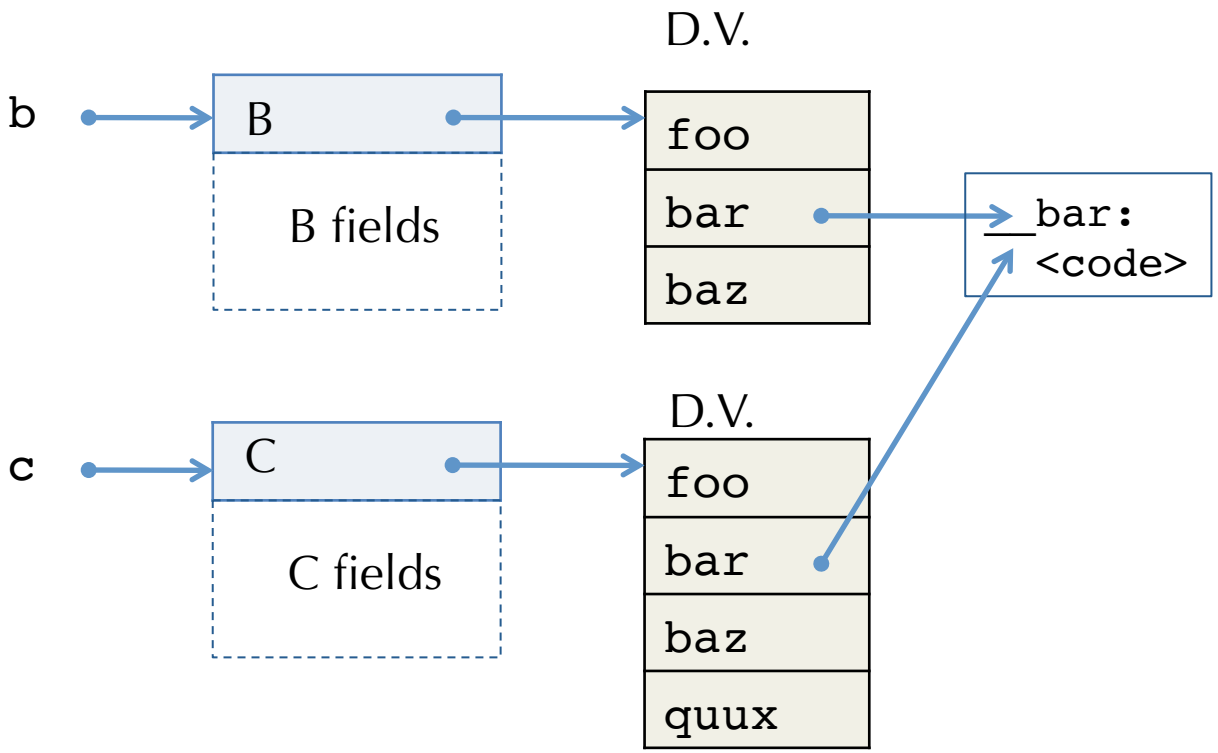
- Smalltalk (ma anche JavaScript) non prevedono un meccanismo per il controllo statico dei tipi
  - l'invocazione di dispatch del metodo **`obj.meth(pars)`** dipende dal flusso di esecuzione
  - ogni classe ha il proprio meccanismo di memorizzazione dei metodi nelle tabelle

Discussione: sharing strutturale

Tutte le istanze di oggetti della stessa classe condividono il Dispatch Vector



Inheritance: sharing del codice



Discussione: C++

- C++ prevede un controllo dei tipi statico degli oggetti
  - offset dei campi degli oggetti (offset_data), la struttura delle vtable è condivisa nella gererachia di ereditarietà
  - offset dei dati e dei metodi sono noti a tempo di compilazione
- Il dispatching "**`obj.mth(pars)`**"

  `obj->mth(pars)` nella notazione C++

  viene pertanto compilato nel codice

  `*(obj->vptr[0])(obj,pars)`

  assumendo che **mth** sia il primo metodo della vtable
- Si noti il passaggio dell'informazione relativa all'oggetto corrente

Dispatching (in dettaglio)

Idea: ogni metodo è caratterizzato da un indice unico
Indice permette di identificare il metodo nella vtable

```
interface A {
  void foo();
}
```

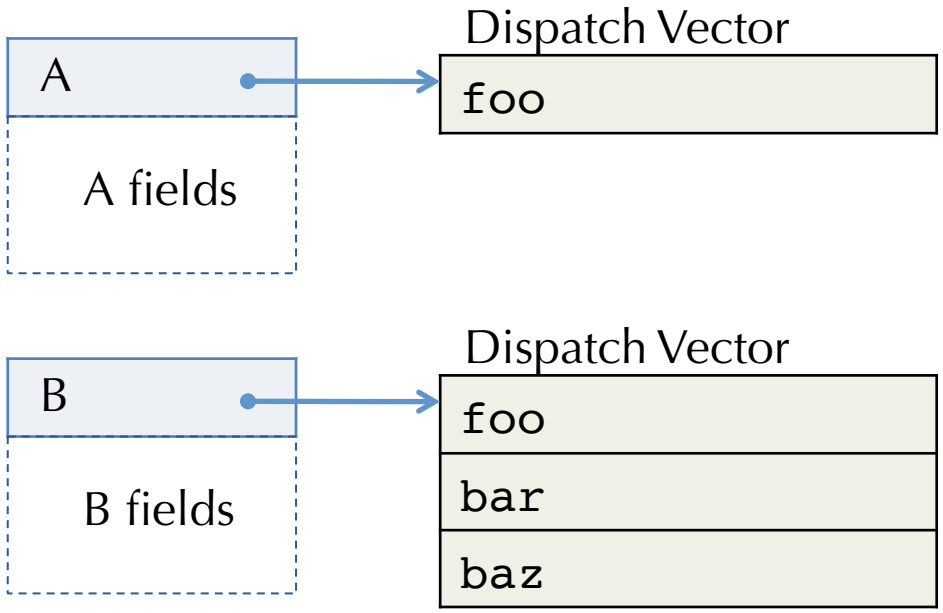
```
interface B extends A {
  void bar(int x);
  void baz();
}
```

```
class C implements B {
  void foo() {…}
  void bar(int x) {…}
  void baz() {…}
  void quux() {…}
}
```

Index
0
1
2
0
1
2
3

Dispatcting

Classi e interface permettono di definire il layout del dispact vector



Metodi statici

- Metodi statici: appartengono alla classe e non possono accedere a variabili di istanza (non hanno il parametro implicito this)
- Run-time: sono implementati esattamente con procedure a top-level
- Sostanzialemente non sono metodi

Ereditarietà Multipla

Ereditarietà multipla

- C++: una classe puo estendere più classi.

```
class A { int m(); }
class B { int m(); }
class C extends A,B {...} // m quale dei due?
```

- Documentazione: “C++, fields and methods can be duplicated when such ambiguity arises”
- Java: una classe può implementare più interface.
- Se le interface contengono lo stesso metodo la classe avrà una sola implementazione

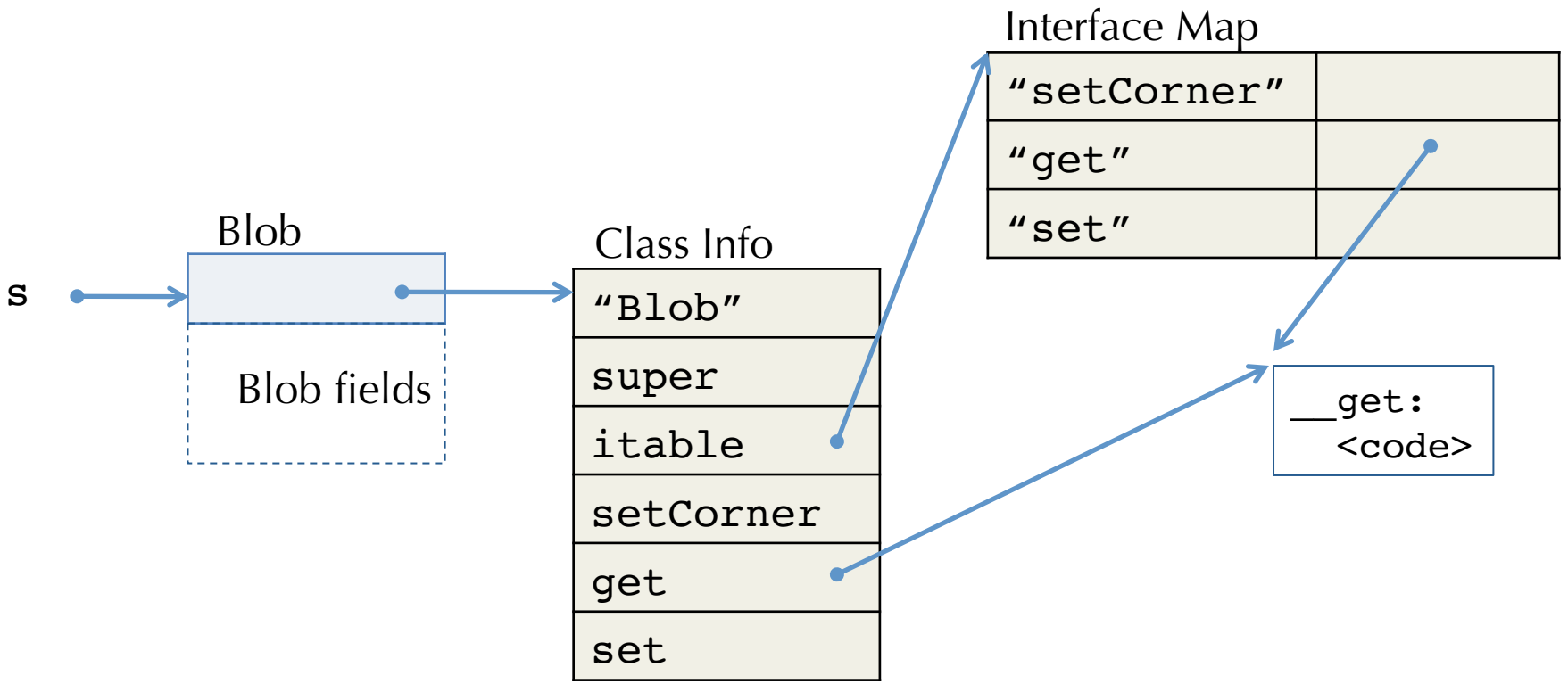
```
interface A { int m(); }
interface B { int m(); }
class C implements A,B {int m() {...}} // solo un cod di m
```

Indici e Dispatch Vector


```
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?
}
```

Soluzione 1

Tabella di supporto Interface Table per associare il codice ai metodi

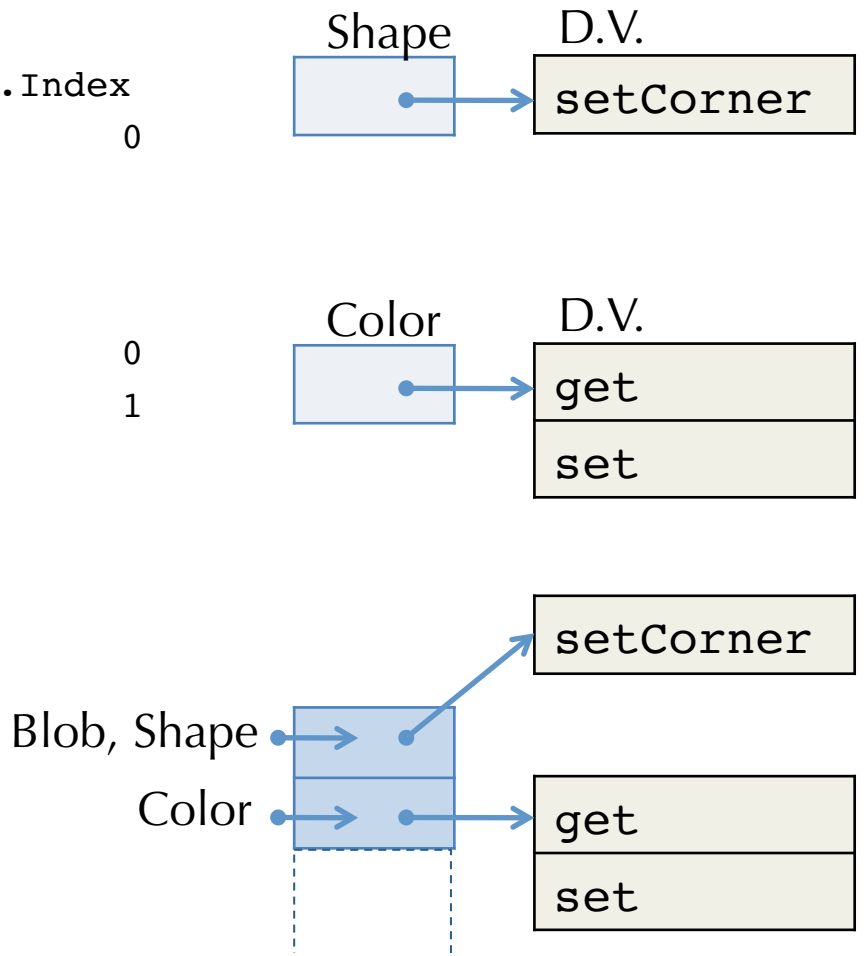


Soluzione 2: hashing per indici


```
interface Shape {                          D.V.Index
  void setCorner(int w, Point p);          hash("setCorner") = 11
}

interface Color {
  float get(int rgb);                      hash("get") = 4
  void set(int rgb, float value);          hash("set") = 7
}

class Blob implements Shape, Color {
  void setCorner(int w, Point p) {…}           11
  float get(int rgb) {…}                       4
  void set(int rgb, float value) {…}           7
}
```

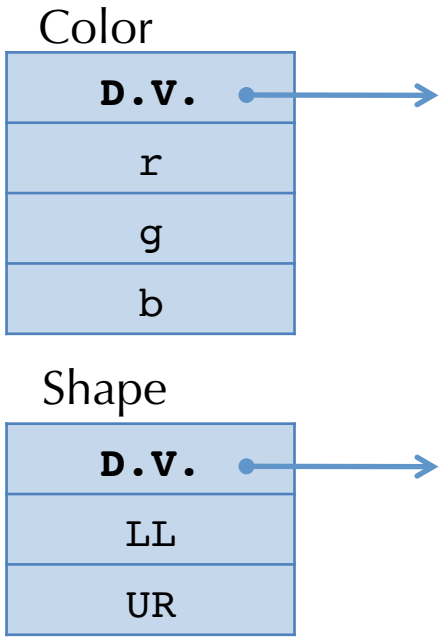
Soluzione 3: duplicare i dispact vector

```
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) {…}
  float get(int rgb) {…}
  void set(int rgb, float value) {…}
}
```



Ereditarietà Multipla (C++)

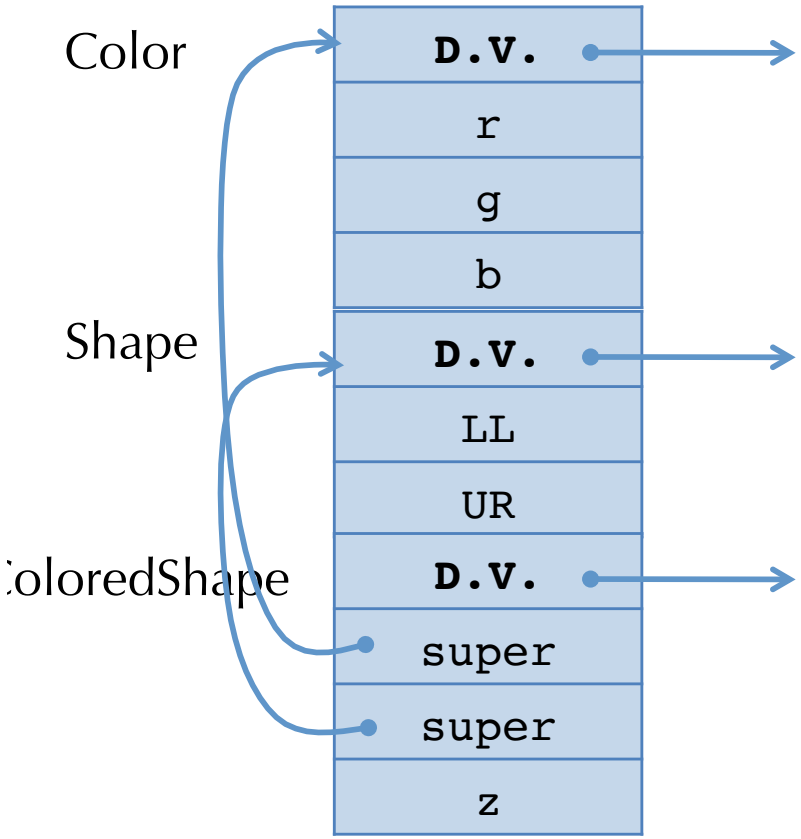
```
class Color {
  float r, g, b; /* offsets: 4,8,12 */
}
class Shape {
  Point LL, UR; /* offsets: 4, 8 */
}
class ColoredShape extends
Color, Shape {
  int z;
}
```



ColoredShape ??

Soluzione C++

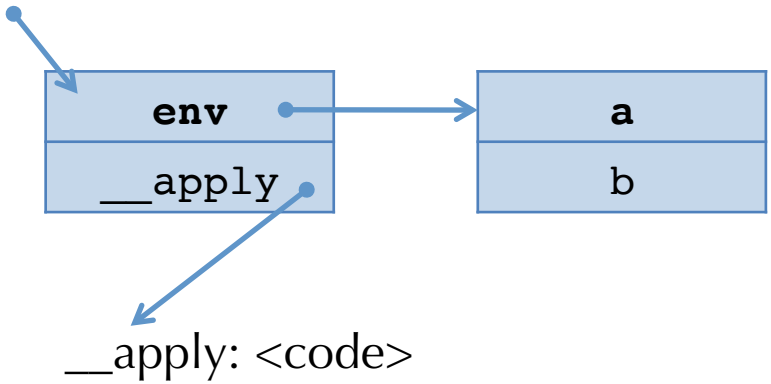
- Aggiungere nella classe i puntatori alle classi padre



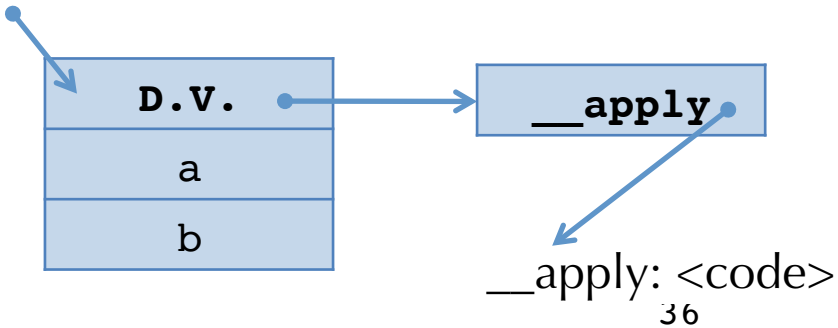
Osservazione sulle chiusure

- Free variables ≈ Fields
- Environment pointer ≈ "this" parameter
- Closure for function: ≈ Instance of this class:

```
fun (x,y) ->
   x + y + a + b
```

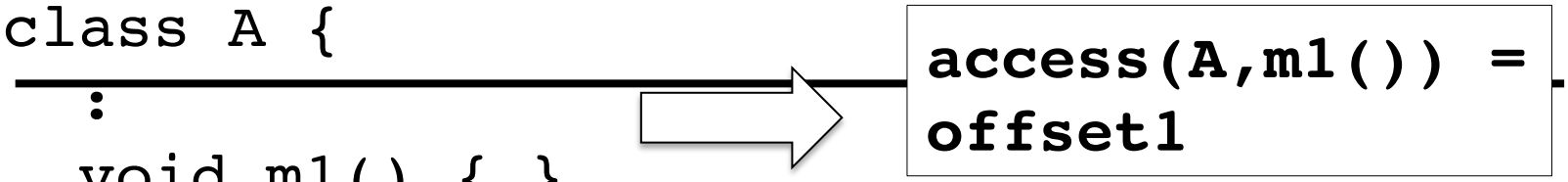


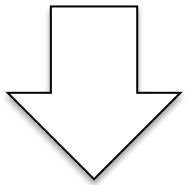

```
class C {
   int a, b;
   int apply(x,y) {
      x + y + a + b
   }
}
```



Compilazione separata

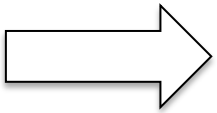
- Compilazione separata di classi (**Java**): la compilazione di una classe produce un codice che la macchina astratta del linguaggio carica dinamicamente (**class loading**) quando il programma in esecuzione effettua un riferimento alla classe
- In presenza di compilazione separata gli offset non possono essere calcolati staticamente a causa di possibili modifiche alla struttura delle classi





Raffinamento della struttura di A

```
class A {
   :
   void ma() {…}
   void mb() {…}
   void m1() {…}
   void m2() {…}
}
```



access(A,m1()) = offset3
[!= offset1]