Paradigma Object-Oriented

Nicola Fanizzi
Dipartimento di Informatica
Università degli Studi di Bari

Linguaggi di Programmazione [010194]
18 mag, 2016

Sommario

1. Introduzione
   - Limiti dell'astrazione dati
2. Concetti fondamentali
   - Oggetti
   - Classi
     - Linguaggi basati su delega
     - Memorizzazione
   - Incapsulamento
   - Sottotipi
     - Ridefinizione di metodi
     - Shadowing
     - Classi astratte
     - Relazione di sottotipo
     - Costruttori
   - Ereditarietà
     - Ereditarietà e visibilità
   - Selezione dinamica dei metodi
     - rel. con l'overloading
     - rel. con lo shadowing
3. Polimorfismo e Generici
   - Polimorfismo di sottotipo
   - Generici in JAVA
     - Segnaposto ? (wildcard)
     - Sottotipi: generici e array
   - Overriding covariante e controvariante
     - Supertipi e viste

Agenda

1 Introduzione
- Limiti dell’astrazione dati

2 Concetti fondamentali

3 Polimorfismo e Generici

Limiti dell'astrazione dati

- Tipo di dato astratto:
  meccanismo nato per garantire incapsulamento e information-hiding chiaro ed efficiente
  - riunisce in un unico costrutto dati e operazioni per manipolarli
- meccanismo troppo *rigido* in caso rispetto a
  - estensione
  - riuso

Limiti dell'astrazione dati – Esempio

```
1  abstype Counter{
2    type Counter = int;
3
4    signature
5      void reset(Counter x);
6      int get(Counter x);
7      void inc(Counter x);
8    operations
9      void reset(Counter x){
10       x = 0;
11     }
12     int get(Counter x){
13       return x;
14     }
15     void inc(Counter x){
16       x = x+1;
17     }
18 }
```

Limiti dell'astrazione dati – Esempio [...cont.]

Rappresentazione (concreta) interi

Operazioni azzeramento, lettura, incremento

Per un contatore arricchito con nuove operazioni
(ad es. contare le invocazioni di `reset`):

1. definire un nuovo ADT simile con l'aggiunta di nuove op.:

```
1  abstype NewCounter1{
2    type NewCounter1 = struct{
3      int c;
4      int num_reset = 0;
5    }
6
7    signature
8      void reset(NewCounter1 x);
9      int get(NewCounter1 x);
10     void inc(NewCounter1 x);
11     int howmany_resets(NewCounter1 x);
```

Limiti dell'astrazione dati – Esempio [...cont.]

```
12
13    operations
14      void reset(NewCounter1 x){
15        x.c = 0;
16        x.num_reset = x.num_reset+1;
17      }
18      int get(NewCounter1 x){
19        return x.c;
20      }
21      void inc(NewCounter1 x){
22        x.c = x.c+1;
23      }
24      // nuova op.
25      int howmany_resets(NewCounter1 x){
26        return x.num_reset;
27      }
28  }
```

Limiti dell'astrazione dati – Esempio [. . . cont.]

- Accettabile per l'incapsulamento
  ma si devono inutilmente ridefinire op. già definite
  (nomi sovraccarichi)
  - 2 copie dello stesso codice
- nessuna relazione tra `Counter` e `NewCounter`:
  - In caso di cambio d'implementazione:
    modifiche alla prima implementazione non si riflettono sulla seconda che va cambiata a mano
  - problemi di ritrovamento dei pezzi di codice da modificare;
  - possibile introduzione di differenze o errori sintattici, ecc.

Limiti dell'astrazione dati – Esempio [...cont.]

2 alternativa: inserire una componente `Counter` nel nuovo ADT

```
1  abstype NewCounter2{
2    type NewCounter2 = struct{
3                            Counter c;
4                            int num_reset = 0;
5                          }
6
7    signature
8      void reset(NewCounter2 x);
9      int get(NewCounter2 x);
10     void inc(NewCounter2 x);
11     int howmany_resets(NewCounter2 x);
12
13   operations
14     void reset(NewCounter2 x){
15       reset(x.c);
16       x.num_reset = x.num_reset+1;
17     }
```

Limiti dell'astrazione dati – Esempio [...cont.]

```
18      int get(NewCounter2 x){
19        return get(x.c);
20      }
21      void inc(NewCounter2 x){
22        inc(x.c);
23      }
24      int howmany_resets(NewCounter2 x){
25        return x.num_reset;
26      }
27  }
```

Limiti dell'astrazione dati – Esempio [. . . cont.]

Osservazioni

- vantaggio:
  - non vanno necessariamente non modificate le operazioni, ma solo richiamate dall'interno di `NewCounter` (con *overloading*)
- svantaggio:
  - obbligo di chiamate esplicite (es. per `get` e `inc`) pur senza modifiche: meglio un meccanismo automatico che permette di ereditare le implementazioni

Limiti dell'astrazione dati – Esempio [. . . cont.]

Problema:
non uniformità di trattamento tra `Counter` e `NewCounter2`

Caso 1: vettore di contatori

- la dichiarazione

  ```
  Counter V[100];
  ```

  non consente anche contatori `NewCounter2` (tipi distinti)
  - analogamente dichiarando `NewCounter2` come tipo base
- Soluzione: Si può ammettere una forma di compatibilità: tutte le op. di `Counter` sono possibili su `NewCounter2`
  - Se `NewCounter2` compatibile con `Counter` un vettore dichiarato `Counter V[100]` potrebbe contenere istanze di entrambi i tipi

Limiti dell'astrazione dati – Esempio [...cont.]

Caso 2: azzeramento di tutti i contatori

```
for (int i=1; i<100; i=i+1)
  reset(V[i]);
```

- problema dell'overloading risolto richiamando la `reset` di `Counter`
- stato dei `NewCounter2` nell'array?
  - campi `num_reset` incrementati?
- la compatibilità ha risolto il problema precedente ma ha compromesso l'incapsulamento:
  - operazione usata non corretta: sempre la `reset` di `Counter` non quella di `NewCounter2`
- soluzione statica all'overloading della `reset`

Soluzione migliore decidere dinamicamente quale metodo (`reset`) applicare a seconda del tipo effettivo

Agenda

1. Introduzione
2. Concetti fondamentali
   - Oggetti
   - Classi
   - Incapsulamento
   - Sottotipi
   - Ereditarietà
   - Selezione dinamica dei metodi
3. Polimorfismo e Generici

Concetti fondamentali

ADT consentono l'*incapsulamento* e l'*information hiding* ma risultano troppo rigidi in generale

Cos'altro servirebbe?

- supporto all'ereditarietà dell'implementazione di alcune operazioni a partire da altri costrutti analoghi
- inquadramento in una nozione di compatibilità in termini delle op. ammissibili per i dati costrutti
- selezione dinamica delle operazioni in funzione del tipo effettivo degli argomenti cui vengono applicate

Concetti fondamentali [...cont.]

Tali requisiti sono soddisfatti dal **paradigma object-oriented** e ne rappresentano le sue caratteristiche essenziali

1. Incapsulamento: astrazione
2. Sottotipi: relazione di compatibilità
3. Ereditarietà: riuso delle implementazioni
4. Selezione dinamica delle operazioni

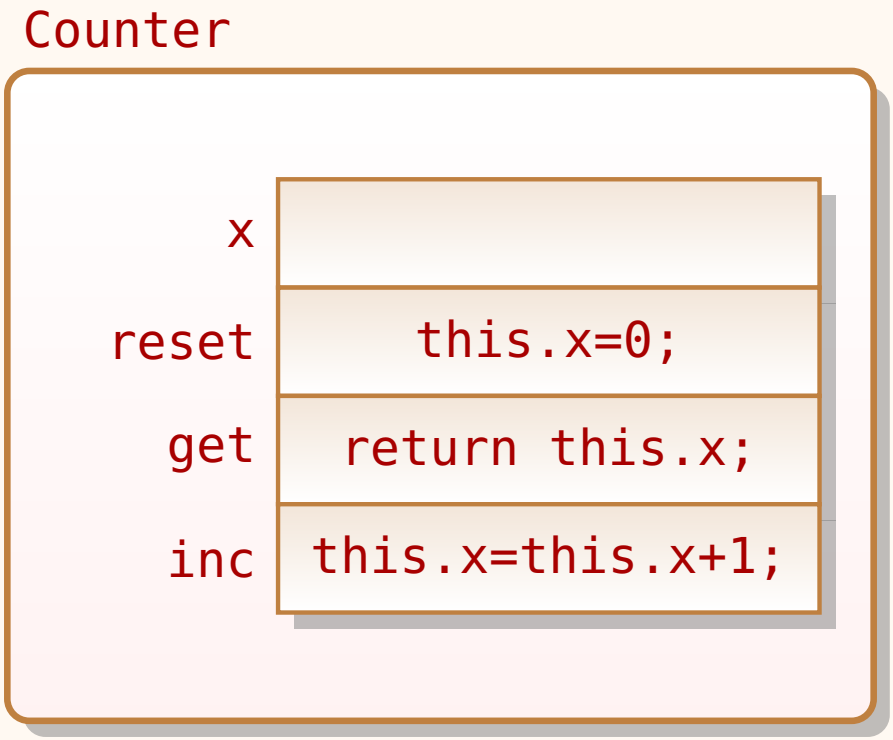
Oggetti

Oggetto: costrutto principale del paradigma:

capsula contenente dati e operazioni su di essi
che fornisce un'interfaccia per l'accessibilità dall'esterno

- in comune con i dati astratti:
  dati insieme alle operazioni
- differenza:
  una var. di tipo astratto rappresenta solo i dati manipolabili attraverso quelle operazioni;
  un oggetto, invece, è un contenitore che incapsula (concettualmente) dati ed operazioni

Oggetti [...cont.]



Oggetti: simili ai record

- campi che corrispondo ai dati (modificabili)
  - es. `x`
- campi che corrispondono alle operazioni ammesse
  - es. `reset`, `get`, `inc`

Oggetti [. . . cont.]

- le operazioni si chiamano **metodi**
  (o campi funzione o funzioni membro)
  e possono accedere ai dati nell'oggetto,
  detti anche variabili istanza (o dati membro o campi)
- esecuzione d'una operazione compiuta mandando all'oggetto un messaggio con il nome del metodo e gli eventuali parametri
  - In JAVA (e C++) la notazione è simile a quella dei record:

```
oggetto.metodo(parametri)
```

Oggetti [. . . cont.]

Nota: l'oggetto che riceve il messaggio costituisce esso stesso un parametro (implicito) per l'invocazione del metodo

- per incrementare un oggetto `Counter o`
  si scriverà `o.inc()`
  - si chiede ad `o` di applicare `inc()` a se stesso

Per i membri-dato stesso meccanismo
(a meno che non siano nascosti dalla capsula)

- per l'oggetto `o`, che ha un campo `v`, si chiede il valore di `v`
  scrivendo `o.v`

Nota: notazione uniforme ma diversa semantica:

- l'invocazione del metodo può comportare, in generale, la selezione dinamica dell'operazione da eseguire,
- l'accesso al dato è statico (salvo alcune eccezioni)

Oggetti [. . . cont.]

Il grado di opacità della capsula viene definito alla creazione

- I campi sono più o meno accessibili direttamente
  - alcune operazioni visibili ovunque,
  - altre solo per alcuni oggetti,
  - altre ancora completamente private (disponibili al solo oggetto)

Nei linguaggi OO costrutti organizzativi per gli oggetti:

- raggruppamento oggetti con la stessa struttura
- il più utilizzato è quello di classe
  - ma ci sono linguaggi che mancano di questa nozione

Classi

Una **classe** è un modello per un insieme di oggetti.

Essa definisce:

- nomi, tipi, visibilità dei suoi dati
- nomi, firme, visibilità e implementazione dei metodi

Ogni oggetto appartiene ad (*almeno*) una classe

Esempio l'oggetto visto in precedenza può appartenere a:

```
1  class Counter{
2
3    private int x;
4
5    public void reset(){
6      x = 0;
7    }
8    public int get(){
9      return x;
10   }
11   public void inc(){
12     x = x+1;
13   }
14
15 }
```

metodi pubblici, campo privato

Classi [. . . cont.]

Oggetti **creati** dinamicamente per istanziazione della loro classe

- allocati secondo la struttura determinata dalla classe

Creazione: op. che varia da linguaggio a linguaggio:

SIMULA classe = *procedura*
restituisce un puntatore ad un RdA con var. locali e definizioni di funzioni

SMALLTALK classe = *oggetto speciale*
schema per la def. dell'implementazione di un insieme di oggetti

C++, JAVA classe = *tipo*
gli oggetti che la istanziano sono valori di quel tipo

Esempio in JAVA:
oggetto contatore

```
Counter c = new Counter();
```

`c` di tipo `Counter` è legato ad un oggetto istanza di `Counter`

Altro oggetto:

```
Counter d = new Counter();
```

A destra dell'assegnazione, due operazioni:

1. creazione degli oggetti (allocazione memoria)
2. inizializzazione chiamata del costruttore di classe: nome + parentesi

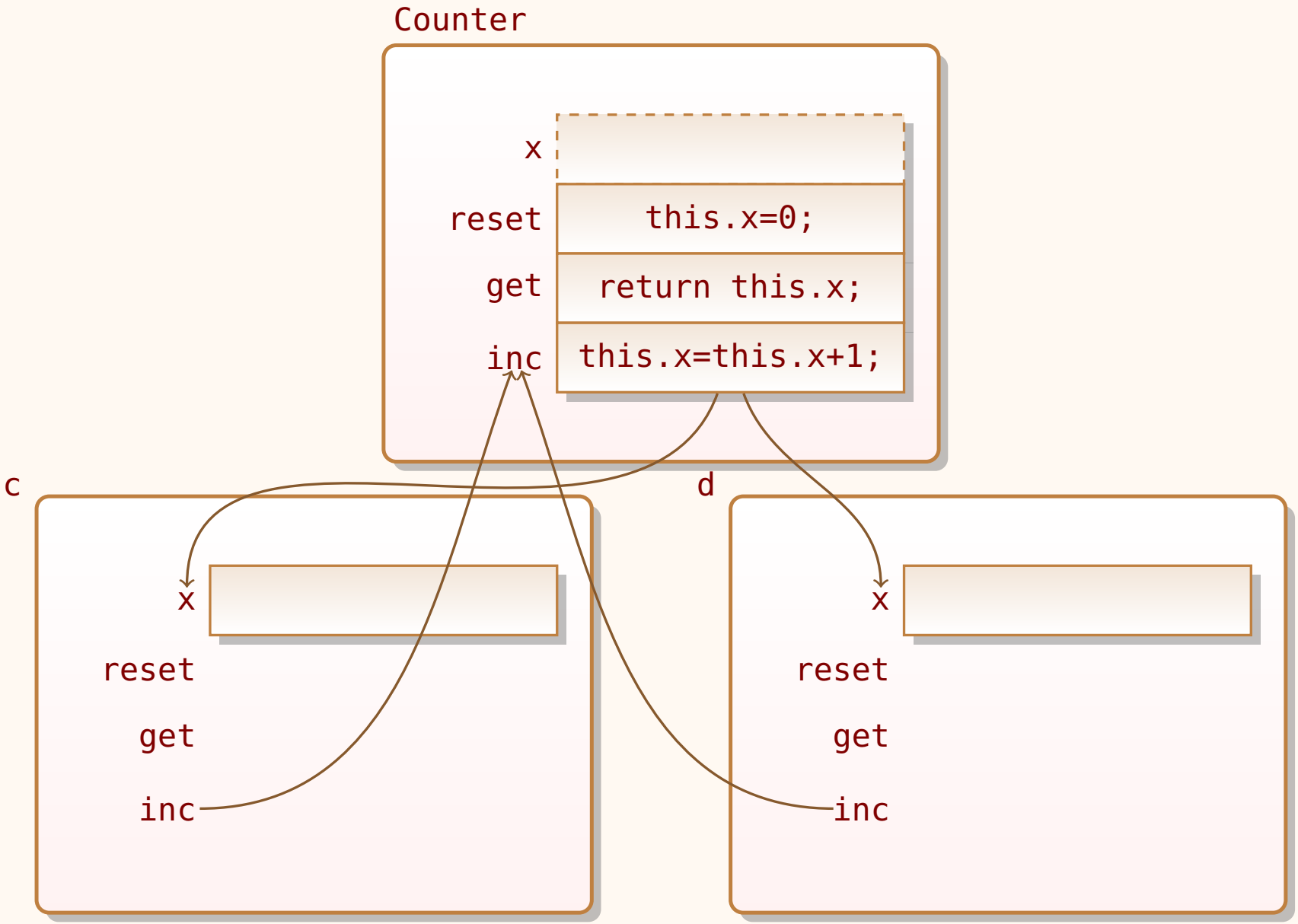
Classi [. . . cont.]

- codice per i metodi condiviso da tutti gli oggetti della classe
- il metodo deve avere accesso ai campi,
  var. diverse per ogni oggetto (nel quale si trovano)
- riferimento ad un campo
  = riferimento implicito all'oggetto che esegue il metodo
- linguisticamente: *nome speciale*, ad es. `self` o **`this`**.
  - Def. equivalente di `inc()`:

```
1     public void inc() {
2       this.x = this.x+1;
3     }
4
```

- nella chiamata della versione con **`this`**,
  il legame tra nome del metodo e codice viene determinato dinamicamente

Classi [...cont.]


Counter
x
reset
this.x=0;
get
return this.x;
inc
this.x=this.x+1;
c
x
reset
get
inc
d
x
reset
get
inc

Classi [. . . cont.]

Alcuni linguaggi permettono di associare variabili e metodi direttamente alle classi e non alle loro istanze:
metodi/variabili statici (o *di classe*)

- variabili: immagazzinate con la classe e non con l'oggetto
- metodi senza **`this`**

Linguaggi basati su delega

Ci sono linguaggi OO senza classi:
il principio organizzativo è la **delega** (o prototipo)

> *un oggetto può delegare ad un altro oggetto (genitore) l'esecuzione di un metodo*

- Esempi: SELF, DYLAN, JAVASCRIPT
- I campi possono contenere:
  - valori (primitivi o oggetti),
  - metodi (codice)
  - riferimenti ad altri oggetti (genitori).
- creazione: implicita o per copia (*clonazione*) di un oggetto *prototipo* (somiglia a una classe)
  - non considerato speciale: oggetto utilizzato come modello
  - alla clonazione si conserva un *riferimento* al prototipo genitore

Linguaggi basati su delega [...cont.]

- quando un oggetto riceve un *messaggio*
  ma non ha il campo nominato passa il messaggio al genitore
  e così via fino a trovare chi possa occuparsene
  - ossia eseguirlo (metodo) / fornire il valore (campo)
- ereditarietà più potente:
  si possono creare oggetti che condividono porzioni di dati
  - nei ling. con le classi si possono solo avere campi **`static`**
- particolarità: il riferimento al genitore può essere modificato *dinamicamente*
  - cambia il comportamento dell'oggetto

Memorizzazione

Uso di memoria dinamica per la *memorizzazione* degli oggetti

Allocazione in mem. dinamica:

heap soluzione più comune

- accesso tramite riferimenti
  - *puntatori* nei linguaggi che li prevedono
  - variabili se i ling. supportano i riferimenti

stack meno comune (es. in C++): come per le var. ordinarie

- ad ogni dichiarazione, implicitamente: *creazione* (e *inizializzazione*)
  - oggetto come valore della variabile

Deallocazione

- in alcuni linguaggi (es. C++) è esplicita
  - uso di metodi distruttori
- nella maggioranza dei casi: garbage collection

Incapsulamento

Incapsulamento e *information hiding*
punti cardinali dell'astrazione dati

- ogni linguaggio OO consente di definire oggetti nascondendo parte di essi (dati/metodi)
- Per ogni classe, due viste:

  privata tutto è visibile: livello di accesso possibile all'interno della classe stessa (per i suoi metodi)

  pubblica solo ciò che è esportato esplicitamente è visibile; come l'interfaccia degli ADT
- con gli oggetti, incapsulamento più *flessibile* ed *estensibile* rispetto agli ADT

Sottotipi

Una classe coincide con l'insieme di oggetti che ne sono istanza: tipo associato alla classe

tipizzati relazione esplicita:
la def. della classe introduce un nuovo tipo che ha per valori le istanze

non tipizzati relazione convenzionale (implicita)
es. SMALLTALK

Si definisce una relazione di compatibilità tra tipi in termini delle operazioni possibili sui loro valori:

> *Il tipo associato alla classe* `T` *è sottotipo di* `S` *quando ogni messaggio compreso da oggetti di* `S` *viene compreso anche da oggetti di* `T`

Sottotipi [. . . cont.]

- se un oggetto viene visto come un record,
  la relazione di sottotipo corrisponde al fatto che `T` è un tipo record contenente (almeno) tutti i campi di `S`
- alcuni campi potrebbero essere *privati*;
  è più preciso dire che:

  > `T` *è sottotipo di* `S` *se*
  > *l'interfaccia di* `S` *è sottoinsieme dell'interfaccia di* `T`

  notare l'*inversione*: il sottotipo ha l'interfaccia più ampia

Alcuni ling. (C++, JAVA) adottano l'*equivalenza per nome* sui tipi in contrasto con la compatibilità completamente *strutturale*

- serve l'esplicita dichiarazione di sottotipo
- def. di sottoclassi o classi derivate
  - di seguito denotata con la parola chiave **`extending`**

Esempio (pseudo-codice)

```
class NamedCounter extending Counter {
  private String name;

  public void set_name(String n){
    name = n;
  }

  public String get_name(){
    return name;
  }
}
```

Ridefinizione di metodi

Oltre ad estendere l'interfaccia, una sottoclasse può ridefinire un metodo della superclasse: method overriding

```
1  class NewCounter extending Counter{
2    private int num_reset = 0;
3
4    public void reset(){
5      x = 0;
6      num_reset = num_reset + 1;
7    }
8
9    public int howmany_resets(){
10     return num_reset;
11   }
12 }
```

Shadowing

Una sottoclasse può ridefinire una var. d'istanza (campo) della superclasse: shadowing

NB: shadowing ≠ overriding a livello implementativo

```
1  class EvenNewCounter extending NewCounter{
2    private int num_reset = 2;
3
4    public void reset(){
5      x = 0;
6      num_reset = num_reset + 2;
7    }
8
9    public int howmany_resets(){
10     return num_reset;
11   }
12 }
```

Shadowing [...cont.]

```
1  class EvenNewCounter extending NewCounter{
2    private int num_reset = 2;
3    public void reset(){
4      x = 0;
5      num_reset = num_reset + 2;
6    }
7    public int howmany_resets(){
8      return num_reset;
9    }
10 }
```

- `num_reset` in `EvenNewCounter` si riferisce alla var. locale (inizializzata a 2) e non a quella di `NewCounter`
- `reset()` su un oggetto `EvenNewCounter` richiama la nuova implementazione che usa il nuovo campo `num_reset`

Classi astratte

In molti linguaggi OO,
classi astratte:
classi che non possono avere istanze perché mancano della specifica dell'implementazione di qualche metodo

- specificano solo la firma di tali metodi (nomi+tipi)
- servono a fornire interfacce
  - possono essere implementate in sottoclassi che definiscono i metodi per i quali manca l'implementazione
  - sono associate a tipi
    quindi rientrano nel meccanismo di estensione delle classi (ereditarietà)

Relazione di sottotipo

La **relazione di sottotipo** tra classi è un *ordinamento parziale*

- NO cicli: non può accadere che `A` sia sottotipo di `B` e `B` sottotipo di `A`, a meno che `A` e `B` coincidano

In molti linguaggi esiste un elemento massimale:

- tipo di cui gli altri tipi sono sottotipi
  - nel seguito denotato con `Object` (come in JAVA)
- un'istanza di `Object` prevede un numero limitato di metodi:
  - clonazione (copia dell'oggetto), test di uguaglianza e pochi altri
  - spesso metodi astratti che necessitano di una definizione per essere usati

Nota In generale, la gerarchia non è un *albero* ma solo un *grafo aciclico orientato*

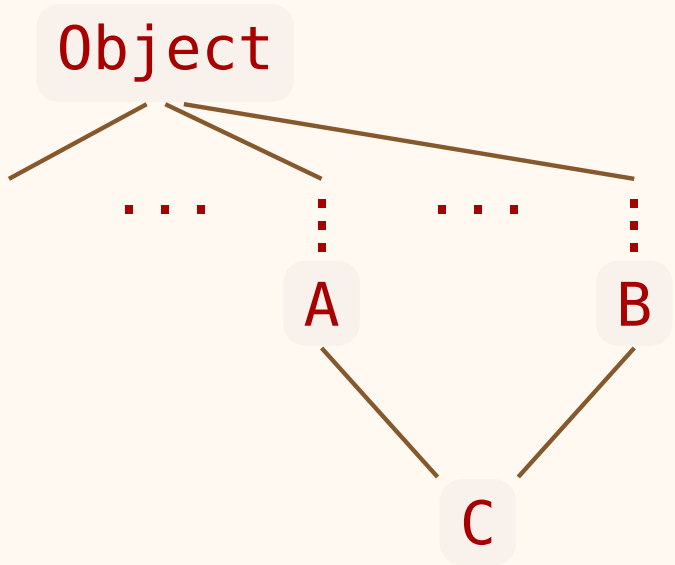
- non è garantito che, dato il tipo `A`, esista un unico tipo `B` che sia l'immediato supertipo

Relazione di sottotipo [...cont.]

Esempio

```
1  abstract class A {
2    public int f();
3  }
4  abstract class B {
5    public int g();
6  }
7  class C extending A,B {
8    private x = 0;
9    public int f() {
10     return x;
11   }
12   public int g() {
13     return x+1;
14   }
15 }
```

C sottotipo di A e di B
senza altri tipi frapposti



Costruttori

La creazione di un oggetto comporta due azioni:

1. *allocazione* della memoria (heap / stack)
2. *inizializzazione* dei dati

L'inizializzazione viene svolta da un **costruttore** della classe:

- codice (metodo) associato alla classe,
  da eseguire *all'atto della creazione* di istanze
- dati coinvolti
  - dichiarati *esplicitamente* nella classe da istanziare
  - dichiarati nelle sue *superclassi*
- spesso sono anche ammessi molteplici costruttori

Costruttori [. . . cont.]

Selezione avviene attraverso il nome del costruttore

- lo stesso della classe (es. in C++ e JAVA)
  - costruttori multipli condividono il nome
    ma si distinguono per tipo e numero dei parametri
    (overloading, risolto staticamente)
    - C++ che consente la creazione implicita di ogg. su stack,
      adotta meccanismi di selezione specifici per ogni caso
- nomi arbitrari dati dal programmatore
  - rimangono sintatticamente distinti dagli altri metodi ordinari

si richiede che ogni creazione (`new`) venga associata ad un particolare costruttore

Concatenazione

come e *quando* inizializzare la parte dell'oggetto che viene ereditata dalla superclasse

- alcuni linguaggi si limitano ad eseguire il costruttore della classe istanziata
  - occorrono chiamate esplicite ai costruttori di superclassi
- altri (C++ e JAVA) garantiscono che, nell'inizializzazione, sia invocato il costruttore della superclasse (chaining) prima d'ogni altra operazione di quello più specifico
  - Problemi: *quale superclasse? quali parametri?*

Ereditarietà

Una sottoclasse che non ridefinisce i campi/metodi della superclasse li eredita: *riusa* l'implementazione

- Es. `NewCounter` eredita da `Counter` il campo `x` e i metodi `inc` e `get` ma non `reset` che viene ridefinito

In generale, l'**ereditarietà** è il meccanismo che permette la definizione di nuovi oggetti/classi riusando quelli/e pre-esistenti

- riuso del codice: modifiche dell'implementazione di metodi di una classe automaticamente disponibili alle sottoclassi

Ereditarietà [. . . cont.]

- rel. di ereditarietà $\neq$ rel. di sottotipo
  - meccanismi *indipendenti* anche se spesso accomunati
    - es. in C++ e JAVA unico costrutto per entrambi

sottotipo relazione tra interfacce di due classi; possibilità di riusare un oggetto in altro contesto

ereditarietà relazione tra implementazioni di due classi; possibilità di riuso del codice che manipola un oggetto

Ereditarietà e visibilità

Tre gradi di visibilità: oltre a quella *privata* e a quella *pubblica*, visibilità di sottoclasse

- una sottoclasse è un particolare client della superclasse: a volte ha accesso a *dati non pubblici*
  - in C++, **`protected`**
- se ha accesso all'*implementazione* il legame è ancora più stretto: ogni modifica della superclasse richiede la modifica delle sottoclassi
  - *pragmatica*: classi dello stesso `package`, come **`protected`** in JAVA
  - un forte *accoppiamento* rende difficili la manutenzione e la modifica

Ereditarietà singola o multipla

Ereditarietà

semplice in certi linguaggi,
una classe può ereditare da una sola superclasse immediata

- La gerarchia è un quindi un albero

multipla altri linguaggi
consentono di ereditare da più superclassi

- la gerarchia è un grafo aciclico orientato

Ereditarietà singola o multipla [...cont.]

Pochi linguaggi supportano l'ereditarietà multipla

- es. C++ ed EIFFEL

problemi teorici (concettuali ed implementativi)
che non hanno soluzioni eleganti

- conflitto (clash) di nomi: *una classe* `C` *estende le classi* `A` *e* `B` *che hanno metodi con la stessa firma*

```
1  class A {
2    int x;
3    int f() {   // condiviso
4      return x;
5    }
6  }
7
8  class B {
9    int y;
10   int f() {   // condiviso
11     return y;
12   }
13 }
14
15 class C extending A, B {
16   int h() {
17     return f(); // quale ?
18   }
19 }
```

Ereditarietà singola o multipla [...cont.]

Soluzioni possibili (insoddisfacenti)

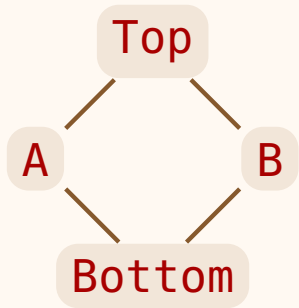
1. proibire i conflitti di nome a livello sintattico;
2. fare risolvere i conflitti al programmatore
   - es. facendo qualificare precisamente ogni riferimento al nome in conflitto
3. adottare una convenzione
   - es. far contare l'ordine della lista di classi da estendere
4. pragmatica:

```
1 class C extending A, B {
2   int f(){
3     return A::f();
4   }
5   int h(){
6     return this.f();
7   }
8 }
```

Ereditarietà singola o multipla [...cont.]

- Diamond problem[1]: *una classe eredita da due superclassi, ognuna delle quali figlia di una stessa superclasse*

```
1  class Top {
2    int w;
3    int f() {
4      return w;
5    }
6  }
7  class A extending Top {
8    int x;
9    int g() {
10     return w+x;
11   }
12 }
13 class B extending Top {
14   int y;
15   int f() {
16     return w+y;
17   }
18   int k() {
19     return y;
20   }
21 }
22 class Bottom extending A, B {
23   int z;
24   int h() {
25     return z;
26   }
27 }
```



[1] Spesso è meglio ricorrere a rel. di sottotipo rispetto a classi astratte

Selezione dinamica dei metodi

La **selezione dinamica** dei metodi (o dispatch) è alla base del paradigma OO: permette la coesistenza di compatibilità di sottotipo ed astrazione

- Un metodo definito per un oggetto può essere ridefinito (*overridden*) in oggetti che appartengono a sottotipi
  - Quando un metodo `m()` viene chiamato su un oggetto `o`, possono esserci tante versioni di `m()` per `o`
- La selezione dell'implementazione di `m()` alla chiamata `o.m(param);` dipende dal tipo dell'oggetto ricevente
- **NB**: il tipo dell'oggetto (*creato*) può differire dal tipo del riferimento all'oggetto (*dichiarato*)
  - nome: info *statica*

Selezione dinamica dei metodi [...cont.]

Esempio in un linguaggio con classi:

```
1  class Counter{
2    private int x;
3    public void reset(){
4      x = 0;
5    }
6    public int get(){
7      return x;
8    }
9    public void inc(){
10     x = x+1;
11   }
12 }
```

```
13 class NewCounter
14   extending Counter{
15   private int num_reset = 0;
16
17   public void reset(){
18     x = 0;
19     num_reset = num_reset + 1;
20   }
21   public int howmany_resets(){
22     return num_reset;
23   }
24 }
```

```
1 NewCounter n = new NewCounter();
2 Counter c;
3 c = n;
4 c.reset();
```

- tipo (statico) di `c` è `Counter` ma esso si riferisce (dinamicamente) ad un'istanza di `NewCounter`
- sarà chiamato `reset()` di `NewCounter`

Selezione dinamica dei metodi [...cont.]

Esempio oggetti di tipo `Counter` e `NewCounter` memorizzati insieme in un vettore che ha il supertipo come tipo base:
`Counter V[100];`

- Un compilatore non può decidere sul tipo di un oggetto a runtime:

```
1 for (int i=1; i<100; i=i+1)
2   V[i].reset();
```

  la selezione dinamica assicura che sia chiamato il metodo corretto per ogni oggetto

Selezione dinamica dei metodi [...cont.]

overloading vs. selezione dinamica
stesso problema: risolvere un caso ambiguo

overloading	soluzione statica basata sui tipi dei nomi coinvolti
sel. dinamica	soluzione a runtime, considera i tipi dal punto di vista dinamico: in base all'oggetto e non al nome

NB non è un caso di polimorfismo: non c'è un solo pezzo di codice

Il meccanismo di associazione dinamica si chiama late binding di `self` (o **`this`**)

Selezione dinamica dei metodi [...cont.]

Esempio

```
1  class A {
2    int a = 0;
3    void f() { g(); }
4    void g() { a=1; }
5  }
```

```
6
7  class B extending A {
8    int b = 0;
9    void g() { b=2; }
10 }
```

Si supponga `o` di tipo `B`
Si vuole chiamare su `o` il metodo `f` ereditato da `A`
Quando `f` chiama `g`: quale versione sarà eseguita?

- l'oggetto che riceve il messaggio è parametro implicito
- la chiamata di `g` in `f` si può scrivere come `this.g()`
- l'oggetto corrente è `o` quindi il metodo invocato è quello di `B`
- così una chiamata come `this.g()` in `f` può riferirsi a metodi non ancora scritti e che saranno disponibili in seguito attraverso la gerarchia delle classi

Selezione dinamica dei metodi [...cont.]

lo shadowing è un meccanismo completamente *statico*

Esempio

```
1  class A {
2    int a = 1;
3    int f() { return -a; }
4  }
5
6  class B extending A {
7    int a = 2;
8    int f() { return a; }
9  }
```

```
10
11 B obj_b = new B();
12 print(obj_b.f());
13 print(obj_b.a);
14
15 A obj_a = obj_b;
16 print(obj_a.f());
17 print(obj_a.a);
```

- le prima e la seconda chiamata a `print` stampano 2
- la terza stampa 2, dato che `f` è ridefinito in `B` (*sel. dinamica*)
  - `obj_b` creato come istanza di `B` quindi ogni accesso, anche attraverso una var. di tipo `A` (come `obj_a`), usa il nuovo metodo (e quindi i campi) di `B`
- l'ultima `print`, invece, stampa 1

Selezione dinamica dei metodi [...cont.]

- non si tratta di selezionare un metodo ma un campo: conta il tipo del riferimento
- `obj_a` è di tipo `A`,
  quindi `obj_a.a`, seleziona il campo di `A`

Osservazione

`obj_b` contiene anche tutti i campi delle superclassi di `B`

Agenda

1. Introduzione
2. Concetti fondamentali
3. Polimorfismo e Generici
   - Polimorfismo di sottotipo
   - Generici in JAVA
   - Overriding covariante e controvariante

Polimorfismo di sottotipo

Polimorfismo:

- ad hoc: overloading
- universale: un valore ha infiniti tipi ottenuti istanziando uno schema di tipo generale

Polimorfismo di sottotipo: un parametro dello schema può essere sostituito con sottotipi di un tipo assegnato

- forma derivante da una nozione di *compatibilità* tra tipi
  - meno generale del polimorfismo universale parametrico
- nella relazione di sottotipo, ogni istanza di classe `A` appartiene a tutte le superclassi di `A`
- interessante specie per il caso dei (parametri dei) metodi
  - passaggio di oggetti appartenenti a sottoclassi delle classi previste

Polimorfismo di sottotipo – esempi

Esempio Si consideri un metodo

```
B foo(A x) { ... }
```

- *compatibilità* di sottotipo rispetto ad `A`:
  - `foo` accetta in input un valore di qualunque sottoclasse di `A`
- il codice di `foo` non deve essere modificato
  polimorfismo di sottotipo implicito
  - limitato ai sottotipi di `A`
- denotando con `<:` la relazione di sottotipo, il tipo di `foo` è:

$$\forall T <: A.T \to B$$

Polimorfismo di sottotipo – esempi [. . . cont.]

Esempio Si consideri la funzione identità su `A`

```
A Ide(A x){ return x; }
```

- dal punto di vista semantico, `Ide` avrebbe il tipo: ∀`T<:A.T->T`
- data la rel. `C<:A` e:

  ```
  C c = new C();
  C cc = Ide(c);
  ```

  un *type-checker statico* segnalerebbe l'errore
- molti linguaggi non sanno riconoscere il tipo di `Ide`:
  manca un *costrutto linguistico* per legare il tipo del risultato a quello dei parametri
- soluzioni:
  - inferenza (es. in ML)
  - downcast: `C cc = (C) Ide(c);`
    evita segnalazioni d'errore a runtime

Polimorfismo di sottotipo – esempi [...cont.]

Esempio stack di `Object`

```
1  class Elem{
2    Object info;
3    Elem prox;
4  }
5
6  class Stack{
7    private Elem top = null;
8
9    boolean isEmpty() {
10     return top==null;
11   }
12
13
14   void push (Object o) {
15     Elem ne = new Elem();
16     ne.info = o;
17     ne.prox = top;
18     top = ne;
19   }
20
21   Object pop() {
22     Object tmp = top.info;
23     top = top.prox;
24     return tmp;
25   }
26 }
```

Polimorfismo di sottotipo – esempi [. . . cont.]

(cont.)
se si assegna un valore `Object` a una var. di un generico tipo `C`:

```
1 C c;
2 Stack s = new Stack();
3 s.push(new C());
4 c = s.pop(); // errore di tipo
```

errore

Serve un cast
`c = (C) s.pop();`
(controllato a runtime)

Generici in JAVA

In JAVA, def. **tipi** (classi e interfacce) e **metodi generici**

- parametri formali di tipo tra parentesi angolari: `<A>`
  - quantificatore universale per i tipi
  - possono essere vincolati entro specifici tipi
  - si può usare anche ? (da leggersi ”sconosciuto”)
    che sta per qualunque tipo
- parametri attuali: classi o tipi array

Esempio stack generico

```
1  class Elem<A> {
2    A info;
3    Elem<A> prox;
4  }
5
6  class Stack<A>{
7
8    private Elem<A> top = null;
9
10   boolean isEmpty() {
11     return top==null;
12   }
13
14   void push (A o) {
15     Elem<A> ne = new Elem<A>();
16     ne.info = o;
17     ne.prox = top;
18     top = ne;
19   }
20
21   A pop() {
22     A tmp = top.info;
23     top = top.prox;
24     return tmp;
25   }
26 }
```

da cui:

```
1 Stack<String> ss = new Stack<String>();
2 Stack<Integer> si = new Stack<Integer>();
3 ss.push(new String("pippo"));
4 String s = ss.pop(); // senza cast dinamico
```

Esempio coppie di elem. generici

```
1  class Coppia<A,B> {
2    private A a;
3    private B b;
4
5    Coppia(A x, B y) {
6      a = x;
7      b = y;
8    }
9    A primo() {
10     return a;
11   }
12
13   B secondo() {
14     return b;
15   }
16 }
```

```
1 Integer i = new Integer(3);
2 String v = new String ("pippo");
3 Coppia<Integer,String> c = new Coppia<Integer,String>(i,v);
4 String w = c.secondo();
```

Metodi Generici

Esempio costruttore di coppie diagonali: componenti identiche

```
1 Coppia<Object,Object> diagonale(Object x){
2   return new Coppia<Object,Object>(x,x);
3 }
4
5 Coppia<Object,Object> co = diagonale(v);
6 Coppia<String,String> cs = diagonale(v); // errore compilatore
```

problematico

meglio:

```
1 <T> Coppia<T,T> diagonale(T x){
2   return new Coppia<T,T>(x,x);
3 }
```

`diagonale()` è di tipo: ∀`T.T->Coppia<T,T>`
usabile anche senza istanziazione:

```
1 Coppia<Integer,Integer> ci = diagonale(new Integer(4));
2 Coppia<String,String> cs = diagonale(new String("pippo"));
```

Generici vincolati: limite ai sottotipi ammessi alla sostituzione del parametro di tipo

Esempio forme

```
1  interface Forma {
2    void disegna();
3  }
4
5  class Cerchio implements Forma{
6    ...
7    public void disegna(){....}
8  }
9
10 class Rombo implements Forma{
11   ...
12   public void disegna(){....}
13 }
```

Data `List<T>`
che appartiene al package `java.util`),
definiamo un metodo per disegnare tutti i suoi oggetti:

```
void disegnaTutto(List<Forma> forme){
  for(Forma f : forme)
      f.disegna();
}
```

corretta ma applicabile solo a parametri `List<Forma>`:

- e.g. `List<Rombo>` non è un sottotipo di `List<Forma>`
  ma solo una diversa istanziazione dello schema

Soluzione parametrica:
`disegnaTutto` di tipo ∀`T<:Forma.List<T> -> void`

```
1 <T extends Forma> void disegnaTutto(List<T> forme){
2   for(Forma f : forme)
3     f.disegna();
4 }
```

da cui:

```
1 List<Rombo> lr = ...;
2 List<Forma> lf = ...;
3 disegnaTutto(lr);
4 disegnaTutto(lf);
```

Esempio trovare il massimo di una lista generica di oggetti confrontabili

```
1 public static <T extends Comparable<T>>
2   T max(List<T> lista)
```

Definita una classe e una lista di oggetti di quella classe:

```
1 class Foo implements Comparable<Object>{...}
2 List<Foo> cf = ....;
```

se si chiama `max(cf)`:

- ogni elemento di `cf` (di tipo `Foo`) è comparabile con qualsiasi oggetto, tra cui ogni `Foo`
- errore:
  `Foo` implementa `Comparable<Object>` e non `Comparable<Foo>`

Serve che `Foo` sia comparabile con uno dei suoi supertipi:

```
1 public static <T extends Comparable<? super T>>
2   T max(List<T> list)
```

Segnaposto ? (wildcard)

? segnaposto per qualunque tipo sconosciuto

Esempio caso precedente:

```
1 void disegnaTutto(List<? extends Forma> forme){
2   for(Forma f : forme)
3     f.disegna();
4 }
```

`List<?>` $\neq$ `List<Object>`

- `List<Object>` non è supertipo di, ad es., `List<Integer>`
- `List<?>` è un supertipo di `List<A>` per ogni `A`

Pragmatica:

- usare ? per una sola occorrenza del parametro
- usare un parametro esplicito per occorrenze multiple

Due ? in uno stesso costrutto devono essere considerati *distinti*

Sottotipi: generici e array

Gerarchia di sottotipo `A <: B`:
preservata dagli array, non preservata dai generici

- `A[] <: B[]`
- `DefPara<A>` e `DefPara<B>` non sono correlati

Motivo le collezioni possono cambiare nel tempo:

- es. se uno `Stack<Integer>` diventasse uno `Stack<Object>` non sarebbero più possibili controlli statici rispetto alla struttura originaria

Sottotipi: generici e array [...cont.]

Esempio stack generico

```
1 Stack<Integer> si = new Stack<Integer>();
2 Stack<Object> so = si; // errato
3
4 so.push(new String("pluto"));
5 Integer i = si.pop(); // pericoloso!
```

Se fosse `Stack<Integer> <: Stack<Object>`
la riga 2 sarebbe corretta: due rif. di tipo diverso allo stesso stack di interi

Il supertipo di `Stack<A>` non è `Stack<Object>` ma `Stack<?>`

Sottotipi: generici e array [...cont.]

Gli **array** preservano i sottotipi: costrutto covariante

Esempio

```
1 Integer[] ai = new Integer[10];
2 Object[] ao = ai; // ok: Integer[] <: Object[]
3 ao[0] = new String("pluto"); // errore a runtime
```

- corretto dal punto di vista statico: per la covarianza
- i controlli dinamici inseriti dal compilatore causano un errore a runtime
  - si tenta di memorizzare una stringa in una var. intera

Sottotipi: generici e array [...cont.]

Array covarianti introdotti in JAVA perché consentono una forma di polimorfismo

Esempio scambio elementi in array

```
1 public swap(Object[] vect) {
2   if (vect.length > 1) {
3     Object temp = vect[0];
4     vect[0] = vect[1];
5     vect[1] = temp;
6   }
7 }
```

`swap` invocabile su qualunque tipo di array per la sua covarianza

Ridefinizione / Overriding

Ridefinizione dei metodi (overriding):

argomenti in generale (JAVA, C++) si richiede che i parametri del metodo siano gli stessi

Esempio Considerata

```
1 class F {
2   C fie (A p) {...}
3 }
```

se si definisce una `G` con un `fie()` con arg. di tipo diverso

```
1 class G extending F {
2   C fie (B p) {...}
3 }
```

non si ha una ridefinizione ma due metodi distinti (*overloaded*)

Ridefinizione / Overriding [...cont.]

risultati JAVA (ver. 5+) e C++ ammettono la sovrascrittura anche per risultati di sottotipi del tipo nel metodo originario

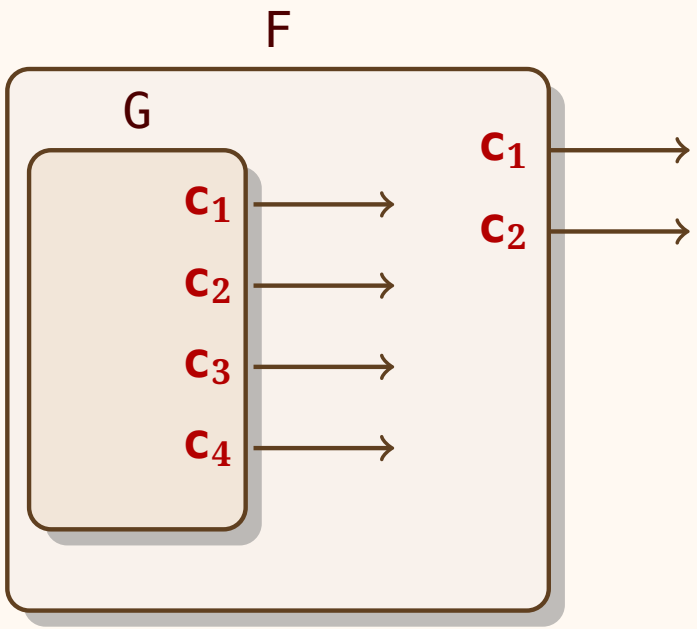
Esempio Supponendo che `D` sia sottotipo di `C` (ossia `D <: C`)

```
1 class E extending F {
2   D fie (A p) {...}
3 }
```

In `E`, sottoclasse di `F`,
`fie()` viene ridefinito rispetto a quello di `F`

Supertipi e viste

Nel polimorfismo di sottotipo,
il tipo di un oggetto è una sua particolare **vista**



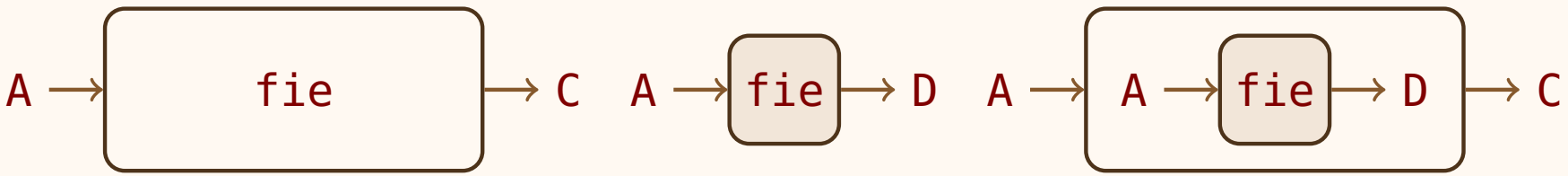
Se `F` è un tipo che prevede i metodi c_1 e c_2
e `G <: F` ottenuto aggiungendo nuovi metodi c_3 e c_4
allora gli oggetti di `G` risponderanno ai metodi di `F` più i nuovi

Overriding covariante e controvariante

Applicando l'idea alla ridefinizione del risultato dei metodi:

- `fie()`, definito in `F` di tipo `A -> C`,
  viene ridefinito in una sottoclasse come di tipo `A -> D`
- corretto quando `D <: C`:
  il nuovo `fie` produce un valore di tipo `D` ma `D <: C`,
  per cui sarà anche un valore di `C`

quindi, sovrascrittura *semanticamente corretta* quando
il metodo è **covariante** rispetto al tipo del risultato

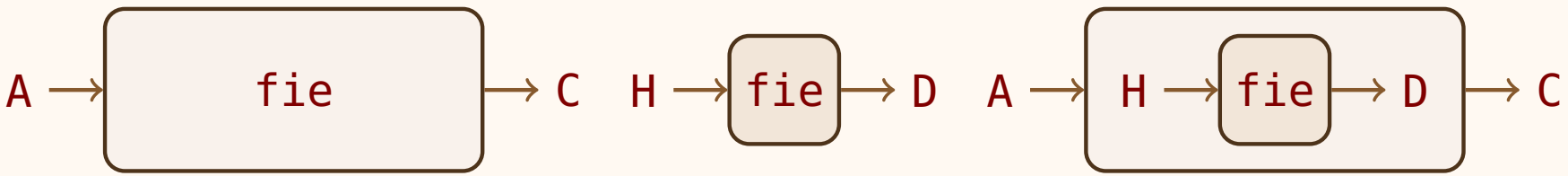


Overriding covariante e controvariante [...cont.]

Ridefinendo il tipo degli argomenti (parametri):

- `fie` ridefinito di tipo `H -> D`
- la relazione tra i tipi `A` e `H` dev'essere `A <: H`

quindi la ridefinizione dev'essere **controvariante** rispetto agli argomenti



Overriding covariante e controvariante [...cont.]

Riassumendo, in generale:

se `S <: S'` e `T' <: T` allora `S -> T <: S' -> T'`

ossia un metodo ridefinito *correttamente* dev'essere

- covariante nel tipo dei risultati e
- controvariante nel tipo degli argomenti

Overriding covariante e controvariante [...cont.]

ridefinizione controvariante (per i parametri) poco diffusa: può risultare *controintuitiva*

Esempio

```
1 class Punto {
2   ...
3   boolean eq(Punto p){...}
4 }
```

```
1 class PuntoColorato extending Punto {
2   ...
3   boolean eq(PuntoColorato p){...}
4 }
```

per la controvarianza `PuntoColorato` non può essere sottoclasse di `Punto` perché il parametro del suo `eq()` è di un sottotipo (non supertipo) del tipo del parametro di `eq()` di `Punto`

Overriding covariante e controvariante [...cont.]

ridefinizione controvariante risp. agli argomenti

- usata solo in EMERALD
- i ling. più diffusi richiedono identici tipi di arg. (C++, JAVA)
  - ad es. metodi binari di confronto, equivalenza
- overriding *covariante* (semanticamente non corretto) in EIFFEL e O_2

Esercizi

Ex.1 Considerando le classi

```
1  class A{
2      int x;
3      int f(){
4          return x;
5      }
6  }
7  class B{
8      int y;
9      int g(){
10         return y;
11     }
12     int h(){
13         return 2*y;
14     }
15 }
16 class C extending A,B{
17     int z;
18     int g(){
19         return x + y + z;
20     }
21     int k(){
22         return z;
23     }
24 }
25 class E{
26     int v;
27     void n(){...}
28 }
29 class D extending E,C{
30     int w;
31     int g(){ return x + y + v; }
32     // ridefinito risp. a C
33     void m(){...}
34 }
```

rappresentare un oggetto istanza di `D`

Ex.2 Date le classi (pseudo-codice)

```
1  abstract class A {
2      int val = 1;
3      int foo (int x);
4  }
5  abstract class B extending A {
6      int val = 2;
7  }
8  class C extending B {
9      int n = 0;
10     int foo (int x) { return x+val+n; }
11 }
12 class D extending C {
13     int n;
14     D(int v) { n=v; }
15     int foo (int x) { return x+val+n; }
16 }
```

Esercizi [. . . cont.]

si consideri il programma

```
1  int u, v, w, z;
2  A a;
3  B b;
4  C c;
5  D d = new D(3);
6  a = d;
7  b = d;
8  c = d;
9  u = a.foo(1);
10 v = b.foo(1);
11 w = c.foo(1);
12 z = d.foo(1);
```

che valori hanno `u`, `v`, `w` e `z` alla fine[2] ?

Ex.3 Date le definizioni (JAVA):

```java
interface A {
    int val = 1; // implicito: static final
    int foo (int x);
}
interface B {
    int z = 1; // implicito: static final
    int fie (int y);
}

class C implements A, B {
    int val = 2;
    int z = 2;
    int n = 0;
    public int foo (int x) { return x+val+n; }
    public int fie (int y) { return z+val+n; }
}
class D extends C {
```

Esercizi [...cont.]

```
18      int val = 3;
19      int z = 3;
20      int n = 3;
21      public int foo (int x) { return x+val+n; }
22      public int fie (int y) { return z+val+n; }
23  }
```

cosa stampa il seguente programma?[3]

```
1 A a;
2 B b;
3 D d = new D();
4 a = d;
5 b = d;
6 System.out.println(a.foo(1));
7 System.out.println(b.fie(1));
8 System.out.println(d.foo(1));
9 System.out.println(d.fie(1));
```

Ex.4 Frammento JAVA corretto?

```
1  class A {
2      int x = 4;
3      int fie (A p) {
4          return p.x;
5      }
6  }
7
8  class B extends A {
9      int y = 6;
10     int fie (B p) {
11         return p.x+p.y;
12     }
13 }
```

Se sì, si può dire che `fie()` viene ridefinito (overridden)?

Esercizi [. . . cont.]

```java
public class binmeth {
    public static void main (String [] args) {
        B b = new B();
        A a = new A();
        int zz = a.fie(a) + b.fie(a) ;
        System.out.print(zz);
    }
}
```

Cosa stampa?

[2] Implementato in Java: 6,6,6,6

[3] 7, 9, 7, 9

Riferimenti

- Gabbrielli & Martini: Linguaggi di Programmazione, McGraw-Hill 2a edizione. Cap. 12