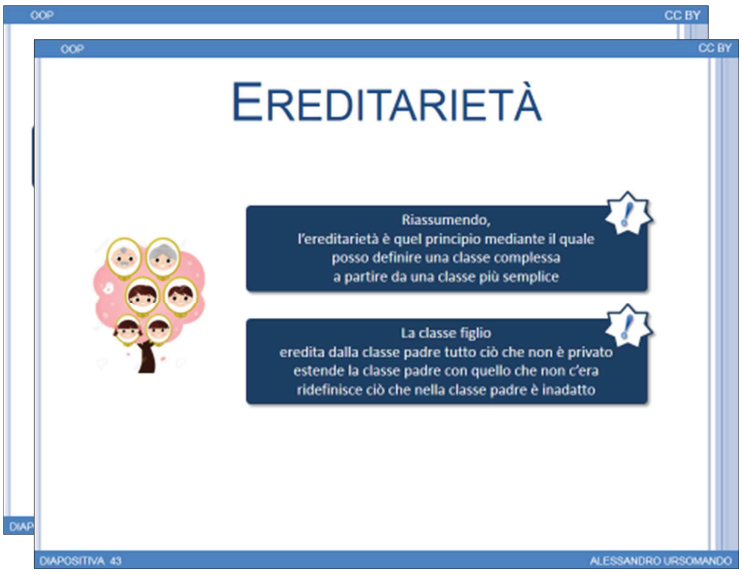
# OOP

## Introduzione al polimorfismo



# LE TRE CARATTERISTICHE FONDAMENTALI DELLA OOP




Abbiamo già visto **l'incapsulamento** che è la prima

Abbiamo già visto **l'ereditarietà** che è la seconda

Manca solo il **polimorfismo.**

# POLIMORFISMO

Il **polimorfismo** è la capacità di un soggetto
di avere comportamenti diversi a seconda del contesto in cui si trova

Il primo tipo di polimorfismo
che vedremo
è il **polimorfismo dei metodi**

Questo tipo di polimorfismo
è molto semplice:
si tratterà solo di fare alcune considerazioni
su dinamiche che abbiamo già visto.





# POLIMORFISMO

Il **polimorfismo** è la capacità di un soggetto
di avere comportamenti diversi a seconda del contesto in cui si trova

Il secondo tipo di polimorfismo
che vedremo
è il **polimorfismo delle classi**

Questo tipo di polimorfismo
è più complesso e la sua comprensione
profonda prevede la conoscenza di altri
concetti (casting e binding).

# OOP

## Polimorfismo dei metodi




# POLIMORFISMO DEI METODI

Abbiamo detto che: il **polimorfismo** è la capacità di un soggetto di avere comportamenti diversi a seconda del contesto in cui si trova

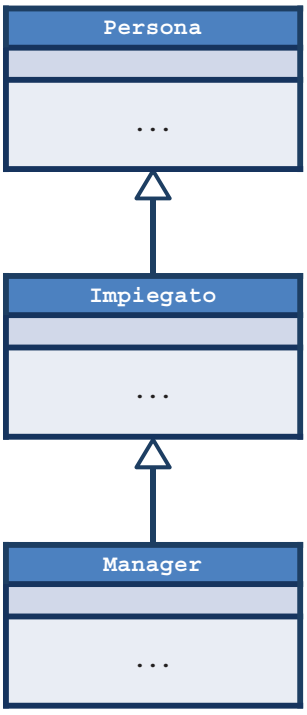
In questo caso il soggetto è il metodo

Osserviamo in che circostanze la stessa invocazione conduce all'esecuzione di blocchi di codice diversi

Le circostanze sono due: l'overriding e l'overloading.


ALESSANDRO URSOMANDO

# POLIMORFISMO DEI METODI MEDIANTE OVERRIDING

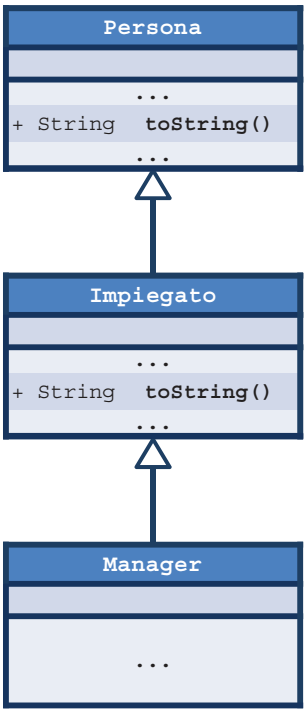



Consideriamo questa gerarchia di classi



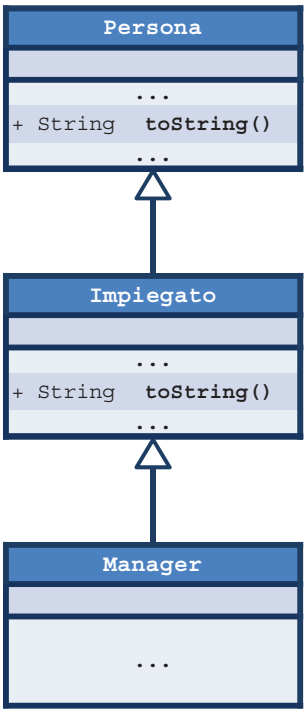



# POLIMORFISMO DEI METODI MEDIANTE OVERRIDING




Consideriamo la **toString** di **Persona**,
l'**override** di questa **toString** da parte di **Impiegato** e
l'**eredità** di questa **toString** da parte di **Manager**

# POLIMORFISMO DEI METODI MEDIANTE OVERRIDING




Consideriamo queste tre istanze

```
Persona p   = new Persona(...);
Impiegato i = new Impiegato(...);
Manager m   = new Manager(...);
```

e queste tre istruzioni

```
System.out.println(p);
System.out.println(i);
System.out.println(m);
```

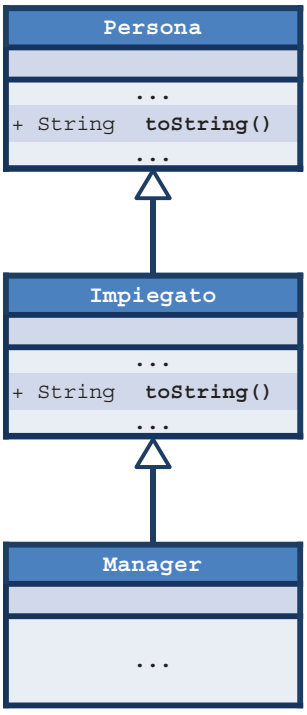
Invoca (implicitamente) la toString di Persona







# POLIMORFISMO DEI METODI MEDIANTE OVERRIDING




Consideriamo queste tre istanze

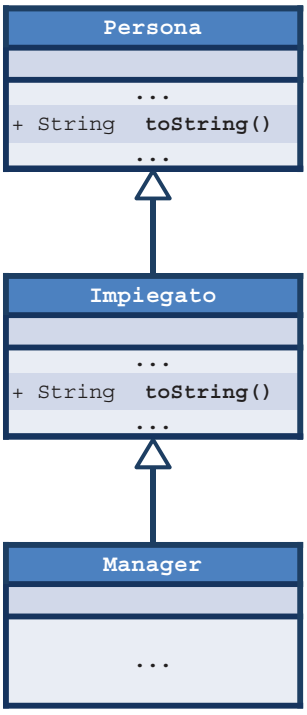
```
Persona p   = new Persona(...);
Impiegato i = new Impiegato(...);
Manager m   = new Manager(...);
```

e queste tre istruzioni

```
System.out.println(p);
System.out.println(i);
System.out.println(m);
```

Invoca (implicitamente) la toString di Impiegato

# POLIMORFISMO DEI METODI MEDIANTE OVERRIDING




Consideriamo queste tre istanze

```
Persona p   = new Persona(...);
Impiegato i = new Impiegato(...);
Manager m   = new Manager(...);
```

e queste tre istruzioni

```
System.out.println(p);
System.out.println(i);
System.out.println(m);
```

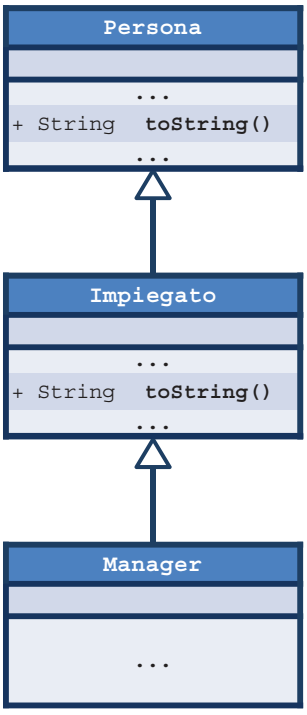
Invoca anch'essa (implicitamente) la toString di Impiegato







# POLIMORFISMO DEI METODI MEDIANTE OVERRIDING




Consideriamo queste tre istanze

```
Persona p   = new Persona(...);
Impiegato i = new Impiegato(...);
Manager m   = new Manager(...);
```

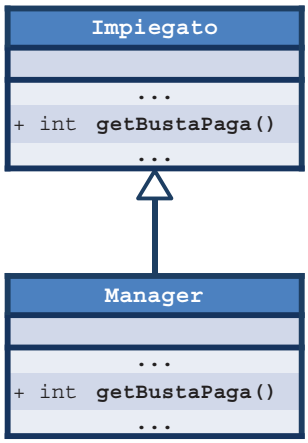
e queste tre istruzioni

```
System.out.println(p);
System.out.println(i);
System.out.println(m);
```

Ogni volta che facciamo overriding di un metodo possiamo dire che quel metodo ha un aspetto polimorfo nella gerarchia delle classi: cambia la classe, cambia il metodo

Facciamo un altro esempio.

# POLIMORFISMO DEI METODI MEDIANTE OVERRIDING




Consideriamo questa gerarchia di classi

Consideriamo queste due istanze

```
Impiegato i = new Impiegato(...);
Manager m   = new Manager(...);
```

e queste due istruzioni

```
int a = i.getBustaPaga();
int b = m.getBustaPaga();
```

Di nuovo:
ogni volta che facciamo overriding di un metodo possiamo dire che quel metodo ha un aspetto polimorfo nella gerarchia delle classi: cambia la classe, cambia il metodo.



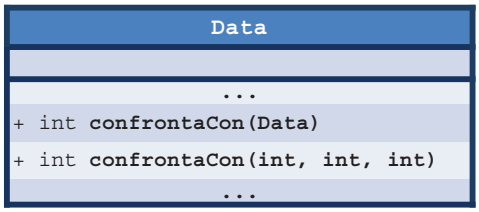




# POLIMORFISMO DEI METODI MEDIANTE OVERLOADING

Sappiamo che abbiamo due tipi di overloading:
overloading orizzontale (nell'ambito di una stessa classe)
overloading verticale (nell'ambito di una gerarchia di classi)

# POLIMORFISMO DEI METODI MEDIANTE OVERLOADING ORIZZONTALE




Consideriamo questa classe

Consideriamo queste due istanze

```
Data d1 = new Data(...);
Data d2 = new Data(...);
```

e queste due istruzioni

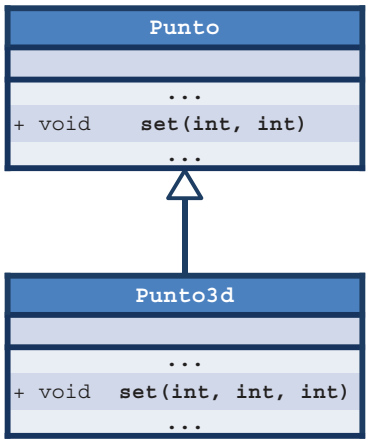
```
int a = d1.confrontaCon(d2);
int b = d1.confrontaCon(19,4,1976);
```

Ogni volta che facciamo overloading orizzontale di un metodo possiamo dire che quel metodo ha un aspetto polimorfo nella classe: cambia l'elenco dei parametri, cambia il metodo.





# POLIMORFISMO DEI METODI MEDIANTE OVERLOADING VERTICALE




Consideriamo questa gerarchia di classi

Consideriamo queste due istanze

```
Punto   p = new Punto(...);
Punto3d q = new Punto3d(...);
```

e queste due istruzioni

```
p.set(12,34);
q.set(-1,81,23);
```

Ogni volta che facciamo overloading verticale di un metodo possiamo dire che quel metodo ha un aspetto polimorfo nella gerarchia delle classi: cambia la classe, cambia il metodo.

# OOP

## Polimorfismo delle classi




# INTRODUZIONE

Ripetiamo ancora una volta che: il **polimorfismo** è la capacità di un soggetto di avere comportamenti diversi a seconda del contesto in cui si trova

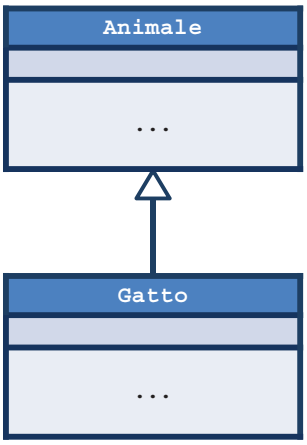
Qui il soggetto è la classe

Stiamo per vedere che è possibile scrivere del codice che chiede a una certa classe di fare una certa cosa ma – al momento in cui si scrive il codice – non è possibile sapere quale classe effettivamente svolgerà il compito

Ok, detta così sembra assurdo ma arriviamoci per gradi. Affrontiamo prima il concetto di casting tra classi.


ALESSANDRO URSOMANDO

# CASTING TRA CLASSI



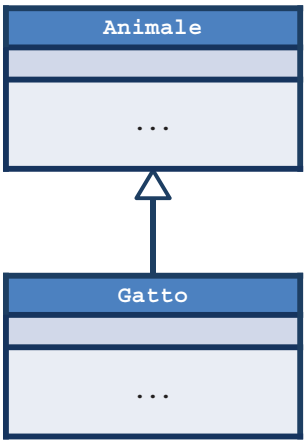

Consideriamo questa gerarchia di classi







# CASTING TRA CLASSI




Consideriamo queste due istanze

```
Animale myAnimale = new Animale(...);
Gatto   myGatto   = new Gatto(...);
```

e questa istruzione

```
myAnimale = myGatto;
```

Poiché la sottoclasse Gatto è figlia della superclasse Animale
l'istruzione è lecita ed è chiamata **casting implicito**.

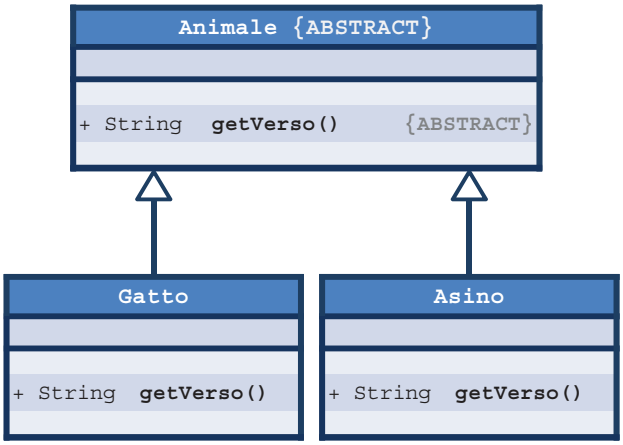
D'altra parte è anche intuitivo: un gatto è un animale!

Lasciamo per un attimo in sospeso il concetto di
**casting esplicito**
e proseguiamo lungo questa strada.

# POLIMORFISMO DELLE CLASSI

Consideriamo questa gerarchia di classi

Consideriamo il metodo astratto getVerso e i suoi override





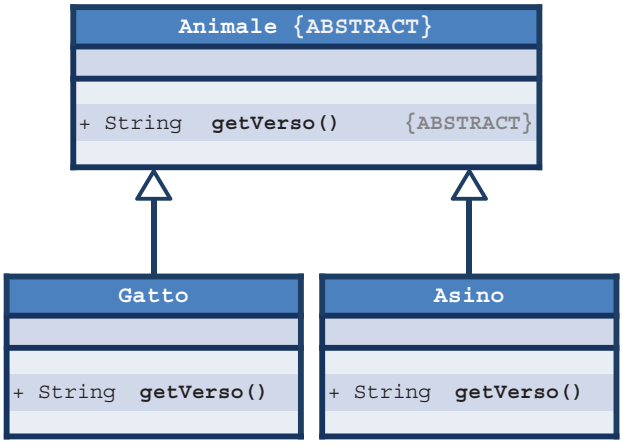





# POLIMORFISMO DELLE CLASSI

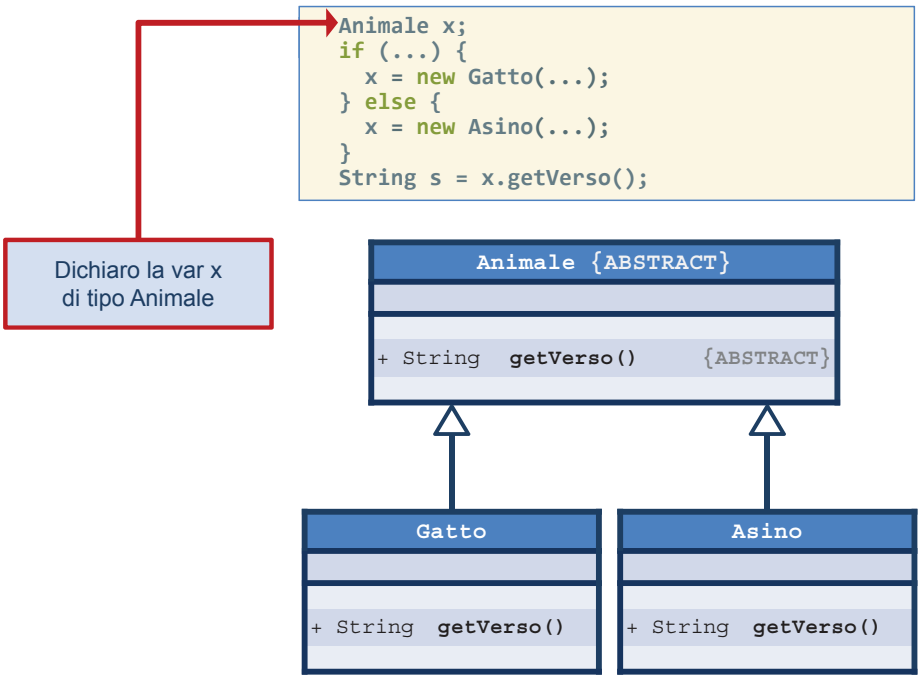
Abbiamo scoperto che è lecita questa istruzione

```
Animale x = new Gatto(...);
```



# POLIMORFISMO DELLE CLASSI

Cosa succede se mettiamo questa istruzione in un blocco condizionale?



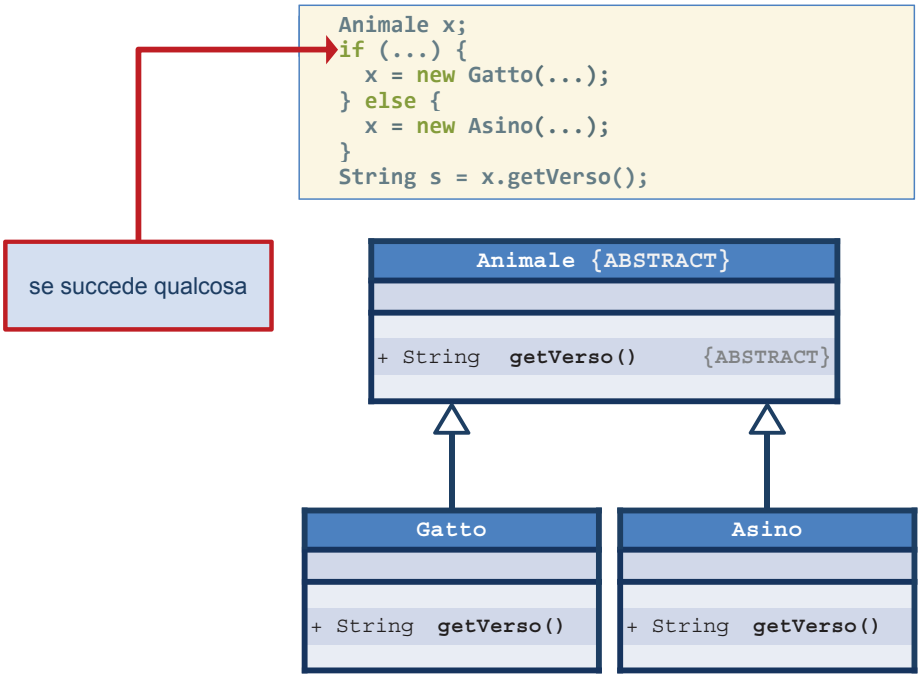






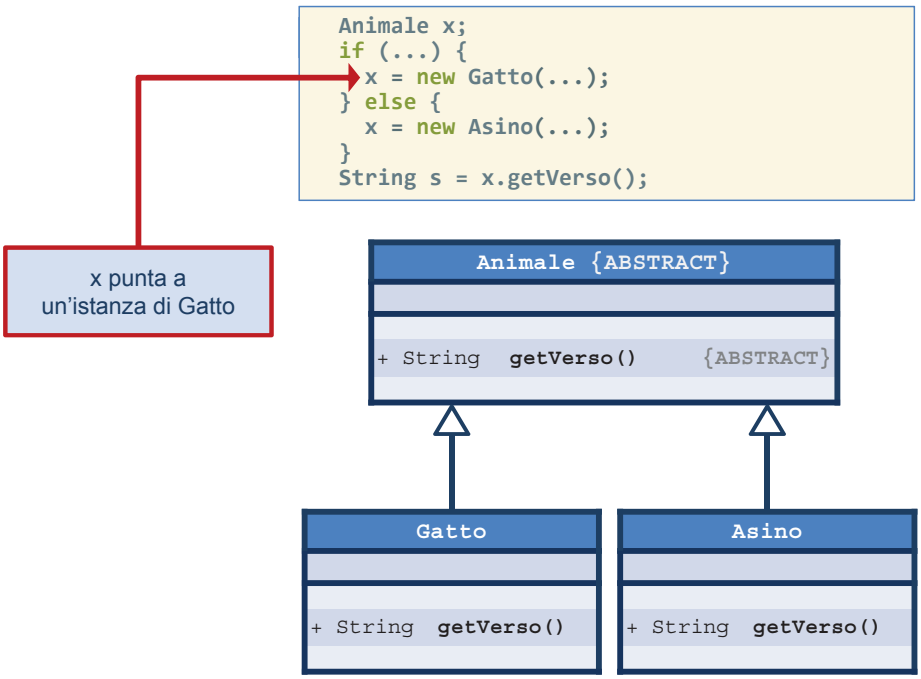

# POLIMORFISMO DELLE CLASSI

Cosa succede se mettiamo questa istruzione in un blocco condizionale?



# POLIMORFISMO DELLE CLASSI

Cosa succede se mettiamo questa istruzione in un blocco condizionale?



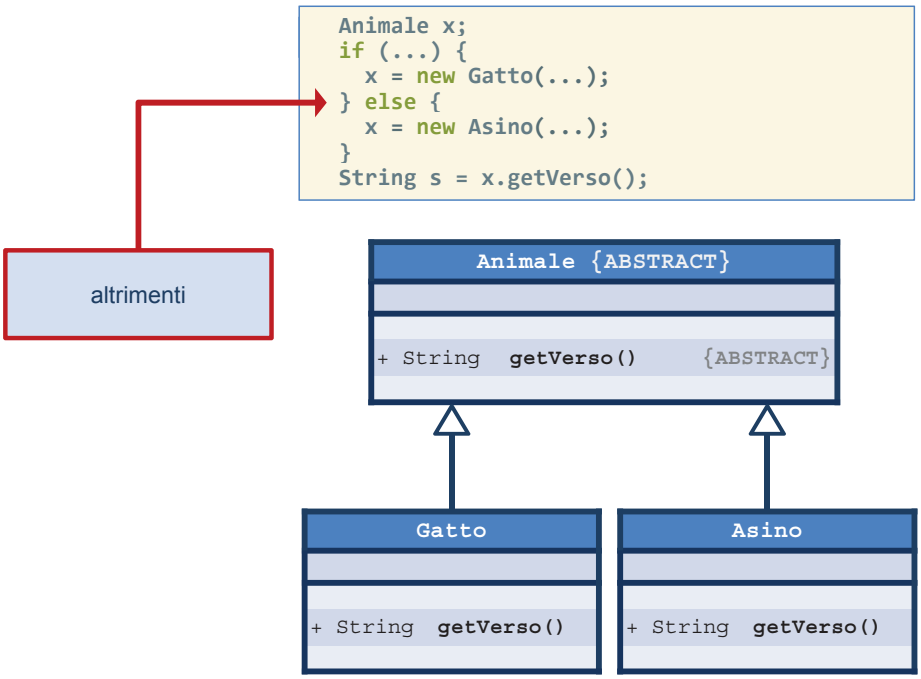






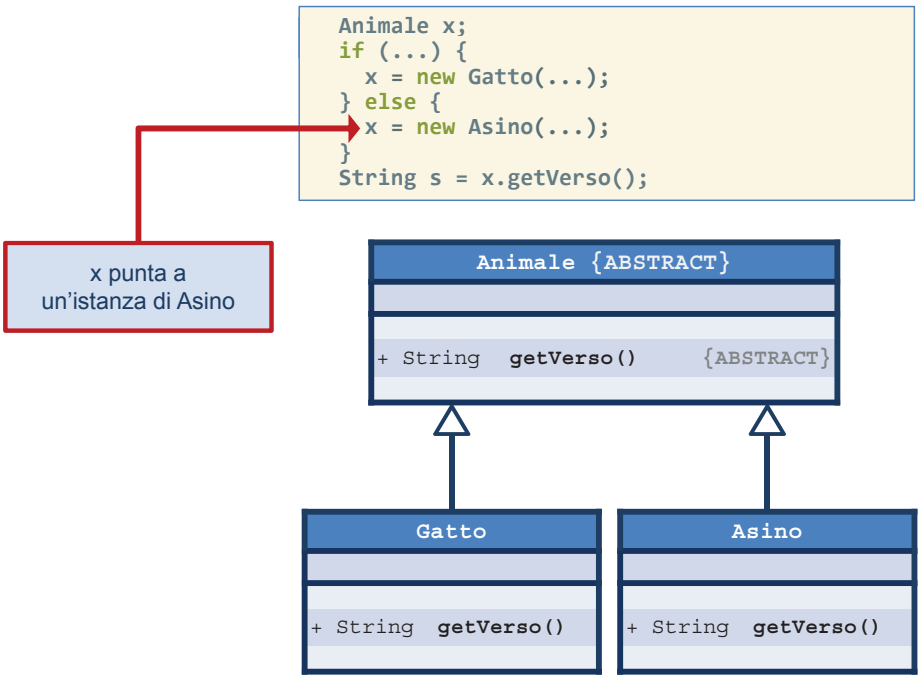

# POLIMORFISMO DELLE CLASSI

Cosa succede se mettiamo questa istruzione in un blocco condizionale?



# POLIMORFISMO DELLE CLASSI

Cosa succede se mettiamo questa istruzione in un blocco condizionale?



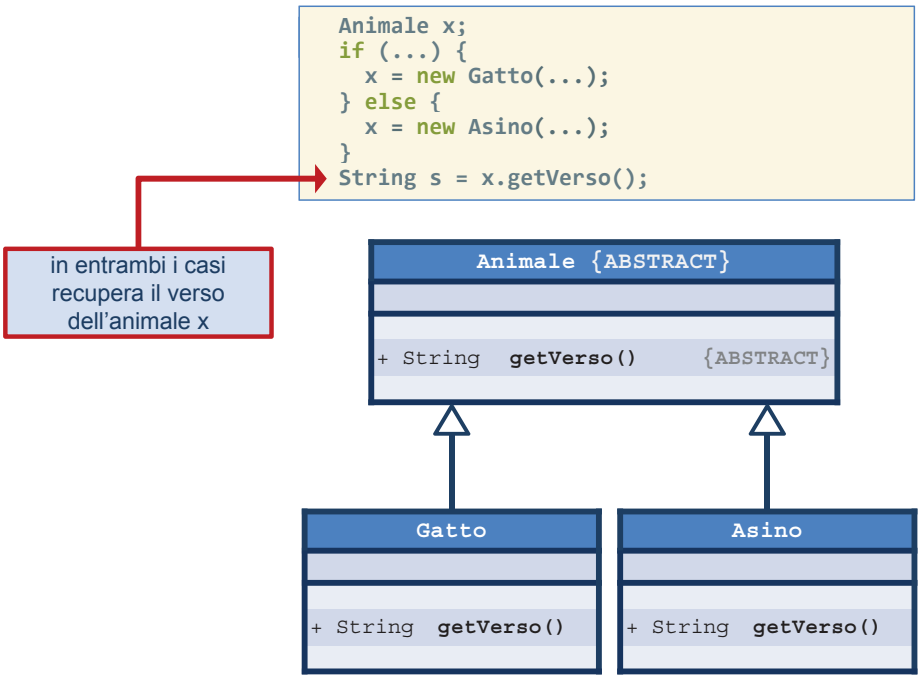







# POLIMORFISMO DELLE CLASSI

Cosa succede se mettiamo questa istruzione in un blocco condizionale?



# POLIMORFISMO DELLE CLASSI

Cosa succede se mettiamo questa istruzione in un blocco condizionale?

```
Animale x;
if (...) {
  x = new Gatto(...);
} else {
  x = new Asino(...);
}
String s = x.getVerso();
```

Succede quindi che a **design-time**
non possiamo sapere se x è un'istanza di Gatto
o di Asino e quindi non possiamo sapere
quale versione di getVerso() si lancerà;
ma la compilazione va a buon fine perché
Animale ha la sua getVerso()

A **run-time** invece,
una volta valutata la condizione,
sapremo se x è un'istanza
di Gatto o di Asino
e potremo eseguire il codice
della getVerso() giusta

L'oggetto x ha un aspetto polimorfo perché posso vederlo come istanza di più classi.






# POLIMORFISMO DELLE CLASSI

Cosa succede se mettiamo questa istruzione in un blocco condizionale?

```
Animale x;
if (...) {
  x = new Gatto(...);
} else {
  x = new Asino(...);
}
String s = x.getVerso();
```

Succede quindi che a **design-time**
non possiamo sapere se x è un'istanza di Gatto
o di Asino e quindi non possiamo sapere
quale versione di getVerso() si lancerà;
ma la compilazione va a buon fine perché
Animale ha la sua getVerso()

A **run-time** invece,
una volta valutata la condizione,
sapremo se x è un'istanza
di Gatto o di Asino
e potremo eseguire il codice
della getVerso() giusta

Per consentire il polimorfismo delle classi
i linguaggi object oriented devono implementare il **binding dinamico**

# COS'È IL BINDING DINAMICO?

La prima cosa che vogliamo dire
è che nasce in contrapposizione al **binding statico**

Il **binding statico** è la metodologia con la quale
i compilatori dei linguaggi procedurali producevano l'eseguibile.







# BINDING STATICO

Prendiamo in considerazione una porzione di codice
di un software scritto con uno pseudolinguaggio procedurale

```
...
```

```
boolean isPrimo (int x) {
   ...
}
```

```
...
```

```
for (int i=0; i<100; i++)
   if (isPrimo(i))
      print(i);
```

```
...
```

Numeriamo tutte le istruzioni

# BINDING STATICO

Prendiamo in considerazione una porzione di codice
di un software scritto con uno pseudolinguaggio procedurale

```
    ...
326  boolean isPrimo (int x) {
327     ...
328  }
    ...
812  for (int i=0; i<100; i++)
813     if (isPrimo(i))
814        print(i);
    ...
```

Numeriamo tutte le istruzioni

questa invocazione di funzione
era mappata
con un'istruzione di basso livello del tipo
`jump 326`

Ah! Allora si che le cose erano semplici.


ALESSANDRO URSOMANDO

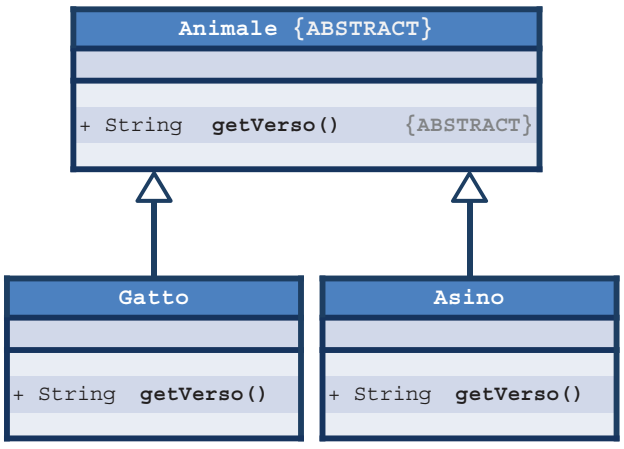



# COS'È IL BINDING DINAMICO?

Adesso, con la programmazione object oriented, è tutto più difficile

```
Animale x;
if (...) {
  x = new Gatto(...);
} else {
  x = new Asino(...);
}
String s = x.getVerso();
```

All'indirizzo di quale getVerso()
dovremmo spedire l'esecuzione?




Non lo sappiamo!

Quando il compilatore Java
produce il bytecode
(quindi siamo a design-time)
non abbiamo nessuna possibilità di saperlo

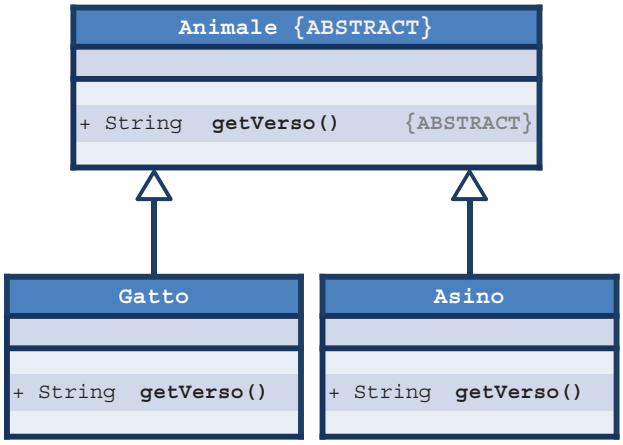
Lo sapremo a run-time


ALESSANDRO URSOMANDO

# BINDING DINAMICO

```
Animale x;
if (...) {
  x = new Gatto(...);
} else {
  x = new Asino(...);
}
String s = x.getVerso();
```

All'indirizzo di quale getVerso() dovremmo spedire l'esecuzione?



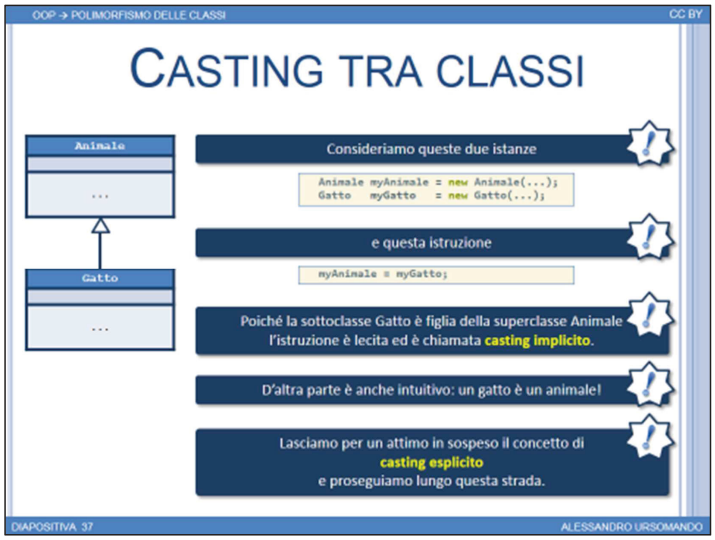

Con il binding dinamico, quindi, il legame tra l'invocazione di un metodo e la sua definizione viene stabilito a run-time.





# .. PAUSA ..

# UN ARGOMENTO IN SOSPESO




Abbiamo iniziato il discorso sul polimorfismo delle classi con il concetto di **casting tra classi**

Abbiamo (ampiamente) affrontato il discorso del **casting implicito** ma non abbiamo detto ancora nulla sul **casting esplicito**

Il **casting esplicito** però è un argomento delicato che (insieme ad altri) ci consente di usare il polimorfismo in modo più spinto



# OOP

## Polimorfismo avanzato

# POLIMORFISMO AVANZATO

Oltre al **casting esplicito**,
dobbiamo conoscere il concetto di **collezione eterogenea**,
il funzionamento del costrutto **instanceof**
e l'uso del metodo **getClass** (ereditato da Object).

Affrontiamo ciascuno di questi concetti singolarmente.

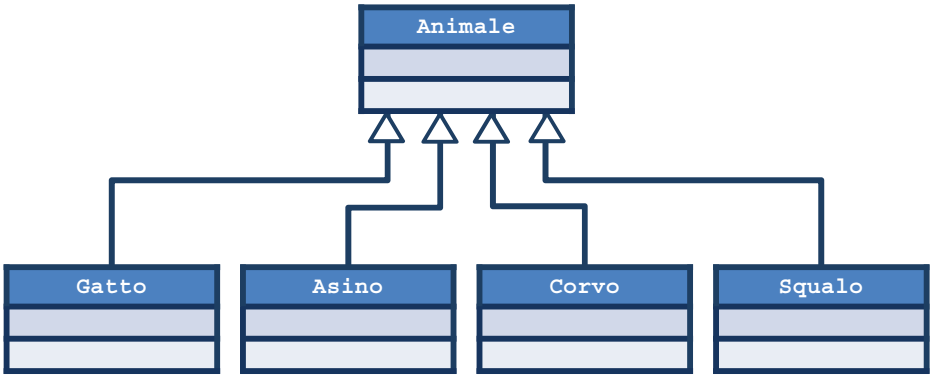




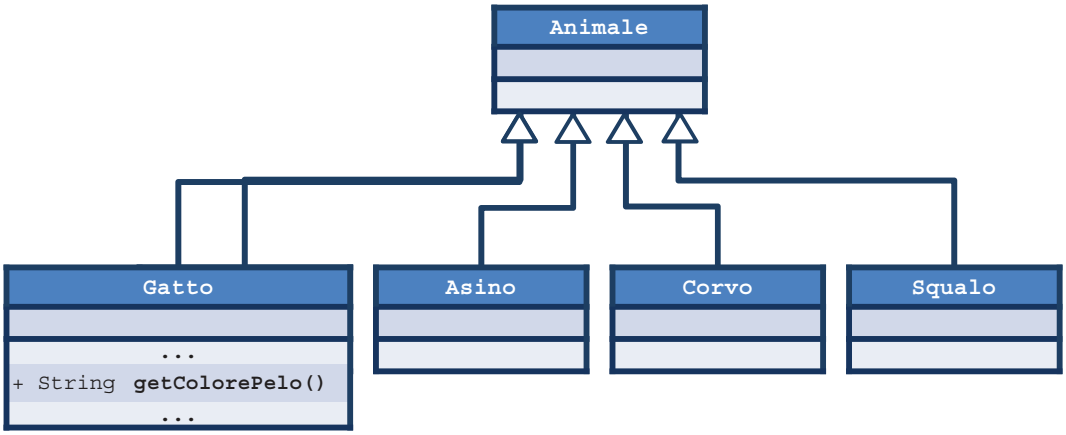


# POLIMORFISMO AVANZATO

Consideriamo questa gerarchia di classi.



# POLIMORFISMO AVANZATO







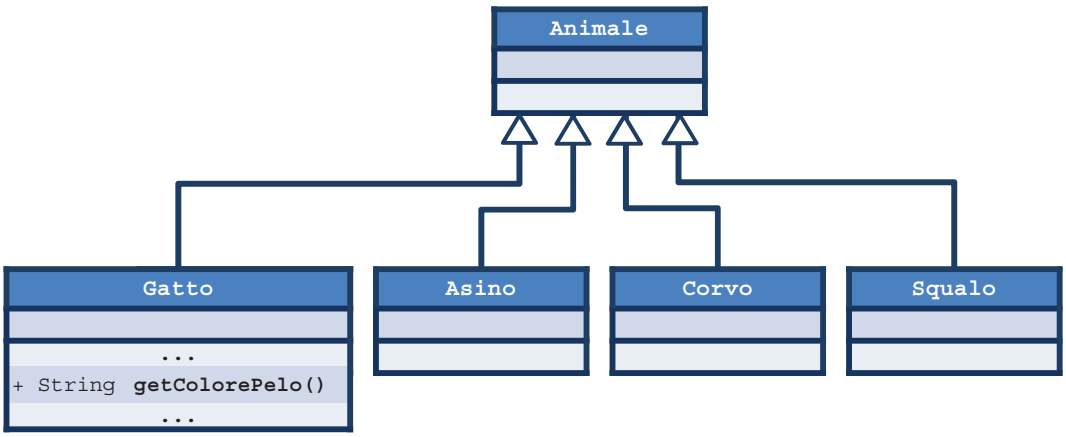

# CAST ESPLICITO

Devo fare un cast esplicito quando ho una istanza di una superclasse che voglio fare diventare un'istanza di una sottoclasse

CASTING ESPLICITO

```
Animale myAnimale = new Gatto(...);
...
Gatto myGatto = (Gatto)myAnimale;
String s = myGatto.getColorePelo();
```

Supponiamo di voler invocare il metodo getColorePelo()

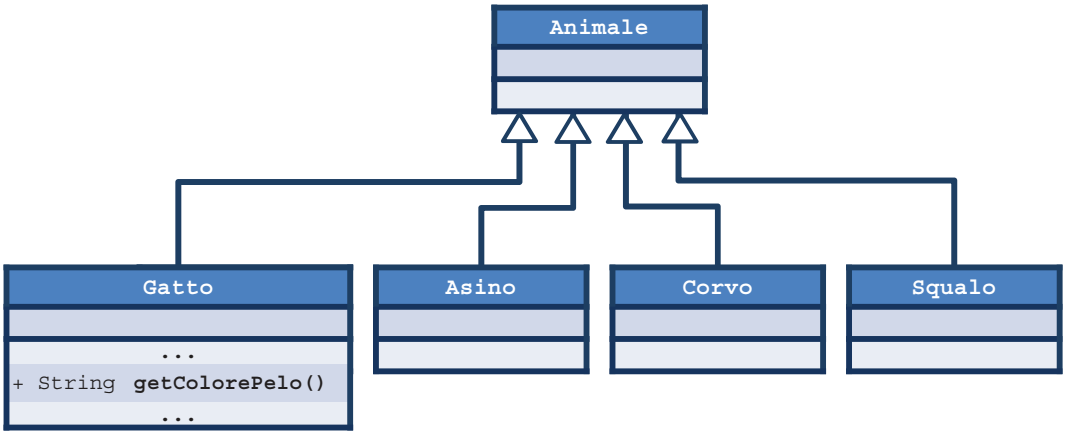


# COLLEZIONI ETEROGENEE

Un vettore dichiarato come collezione di oggetti di una certa superclasse, può evidentemente contenere per ciascuna cella istanze di sottoclassi diverse

COLLEZIONI ETEROGENEE

```
Animale[] animali = new Animale[100];
...
animali[32] = new Gatto(...);
animali[67] = new Asino(...);
animali[71] = new Corvo(...);
```





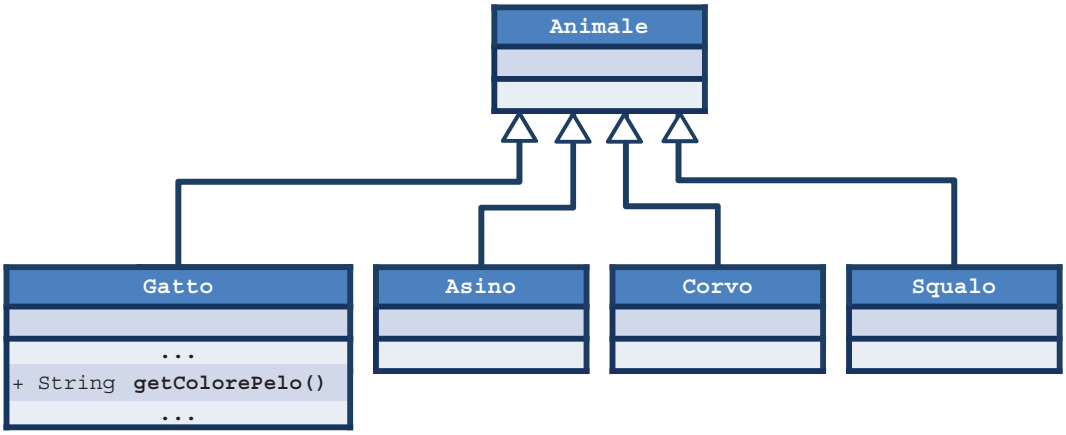



# INSTANCEOF

Per verificare se un certo oggetto è istanza di una certa classe posso usare il costrutto instanceof

INSTANCEOF

```
Animale[] animali = new Animale[100];
...
if (animali[i] instanceof Gatto) {
   Gatto myGatto = (Gatto)animali[i];
   String s = myGatto.getColorePelo();
}
```

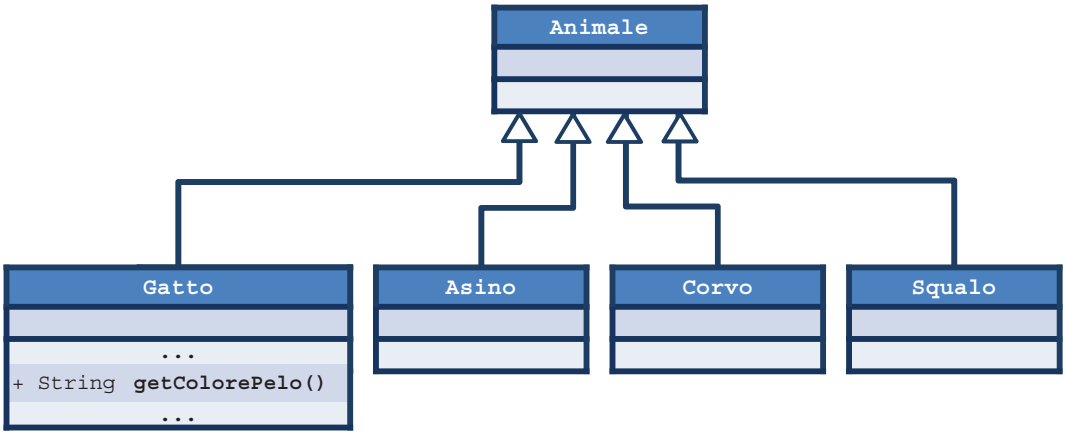


# GETCLASS

Per conoscere la classe di un oggetto posso usare il metodo (ereditato da Object) getClass

GETCLASS

```
Animale[] a = new Animale[100];
...
if (a[i].getClass() == a[j].getClass) {
   ...
}
```










# POLIMORFISMO

Com'è intuibile questi strumenti sono quelli che ci permettono di fare veri e propri giochi di prestigio!

Vale quindi quanto detto prima che casting espliciti, collezioni eterogenee, l'uso di istanceof e del metodo getClass sono per un utilizzo spinto del polimorfismo

Ma che oltre a questo polimorfismo delle classi spinto c'è un polimorfismo delle classi più semplice, il polimorfismo dei metodi per overriding, il polimorfismo dei metodi per overlaoding orizzontale, e il polimorfismo dei metodi per overloading verticale.

In tutti questi casi vi è sempre **la capacità di un soggetto di avere comportamenti diversi a seconda del contesto in cui si trova**