

Classi astratte e progettazione OOP

Esempio: l'enciclopedia degli animali

Ereditarietà, polimorfismo e altri sporchi trucchi

- Facciamo un esempio che ci permette di comprendere come vengono utilizzate le classi astratte e le gerarchie di ereditarietà
- L'esempio mostra alcune tecniche tipiche della progettazione object-oriented e fa largamente uso di **ereditarietà** (sia **subtyping** che **subclassing**) e **polimorfismo**

Enciclopedia degli animali

- Vogliamo realizzare un'enciclopedia, costituita da una serie di schede di animali, a cui possiamo chiedere i dati relativi ad un animale di cui conosciamo il nome.
- I dati sono:
 - **Come si muove:** cammina, vola, nuota
 - **Come si riproduce:** oviparo, viviparo
 - **Da che cosa è ricoperto:** peli, piume, squame
- Per semplicità prendiamo in considerazione solo mammiferi, pesci e uccelli.
- Ogni scheda deve poter fornire la propria descrizione sotto forma di una stringa di questo tipo:
"Nome: Cavallo – Caratteristiche: cammina, è ricoperto di peli, è viviparo"

Un primo tentativo di soluzione

- Una soluzione semplice è quella di creare un'unica classe – Animale – e nel costruttore indichiamo i dati richiesti: nome, riproduzione, movimento, ambiente

```
Animale a1, a2, a3, a4;
```

```
a1 = new Animale("Cavallo", "Viviparo", "Cammina", "Peli");
```

```
a2 = new Animale("Cane", "Viviparo", "Cammina", "Peli");
```

```
a3 = new Animale("Rondine", "Oviparo", "Volta", "Piume");
```

- Questa soluzione ha alcuni difetti:
 - **E' poco estensibile**: se decidiamo di inserire una nuova informazione nelle schede, dobbiamo andare a cambiare il costruttore e tutti i punti in cui vengono create le istanze
 - **Modella male la realtà**: non tiene conto che tutti i mammiferi sono vivipari, tutti i pesci e gli uccelli ovipari

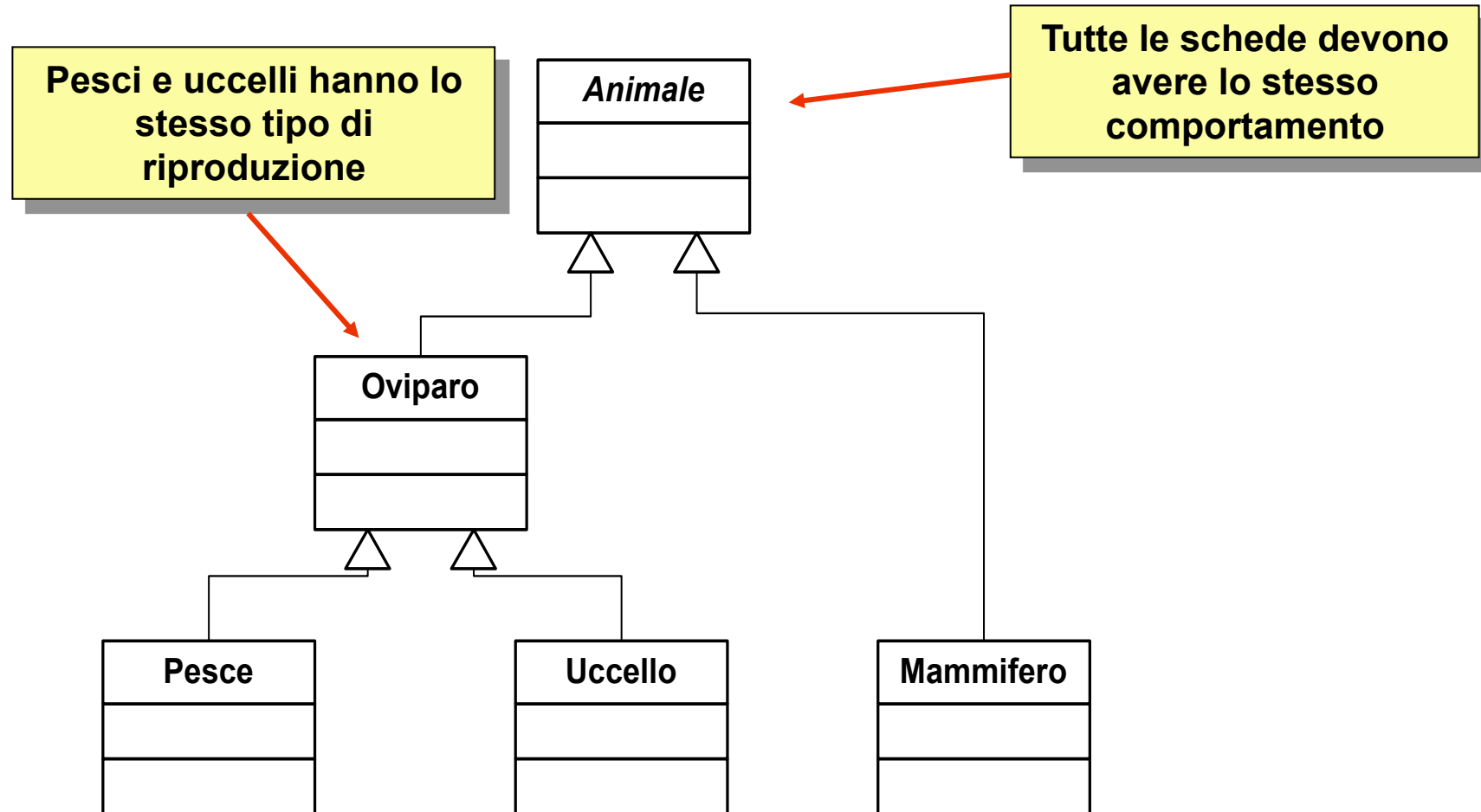
Gerarchie e fattorizzazione

- Il modo corretto di procedere è quello di utilizzare una gerarchia basata sull'ereditarietà
- Si cerca di **fattorizzare** quanti più elementi possibili per evitare duplicazioni nella descrizione
- E' un meccanismo simile al raccoglimento a fattor comune dell'algebra:

$$ab + ac = a(b+c)$$

- Nella programmazione OO le cose funzionano così:
- **Se si individuano caratteristiche comuni a due o più classi si crea una classe base dotata di queste caratteristiche da cui le classi in questione ereditano**

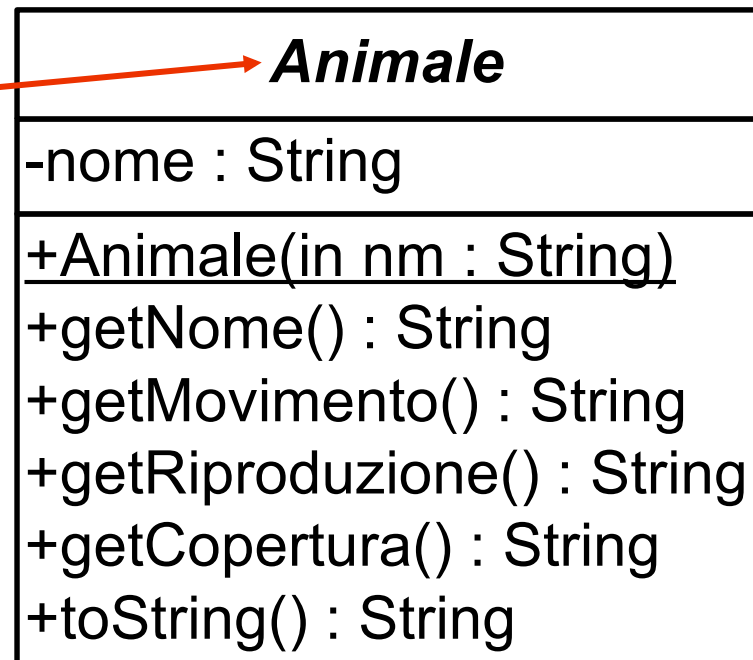
Applichiamo la fattorizzazione



La classe Animale – Una base comune

- Le schede devono essere intercambiabili fra di loro
- Le facciamo quindi discendere da una classe comune che definisce tutti i metodi che ci servono
- Sarà una **classe astratta** perché non siamo in grado di fornire un'implementazione per tutti i metodi

Il nome è in
corsivo perché è
una classe astratta



La classe Animale – Definizione astratta

- Potremmo definirla come classe **completamente astratta**:

```
public abstract class Animale
{
    private String nome;
    public abstract String getNome();
    public abstract String getMovimento();
    public abstract String getRiproduzione();
    public abstract String getCopertura();
}
```

- **Attenzione:** non definiamo toString() come metodo astratto
- E' definito in Object e la classe Animale, discendendo da Object come tutte le classi Java, lo eredita

Animale – Fattorizzazione

- In realtà siamo in grado di fornire un'implementazione per il costruttore e i metodi: `getNome()` e `toString()`:

```
public Animale(String nm)
{ nome=nm; }
public String getNome()
{
    return nome;
}
public String toString()
{
    return "Nome: "+getNome()+ " - Caratteristiche: "+
        getMovimento()+ ", "+getRiproduzione()+", "+
        getCopertura();
}
```



Metodi astratti!

- Che trucco è questo? Come possiamo invocare metodi che non sono stati implementati?

Animale – polimorfismo e subtyping

- Stiamo utilizzando polimorfismo e subtyping
- **Subtyping:** anche se astratti i metodi sono comunque definiti in Animale e quindi possono essere invocati
- **Polimorfismo:** nelle classi derivate verranno invocate le implementazioni specifiche dei metodi
- E' una tecnica non facile da “digerire” ma è molto utile: possiamo definire un comportamento complessivo anche se i dettagli non sono stati ancora definiti
- Nel nostro caso siamo in grado di decidere come sarà costituita nel suo complesso la scheda degli animali lasciando alle classi concrete il compito di implementare correttamente i dettagli

La classe Animale: versione definitiva

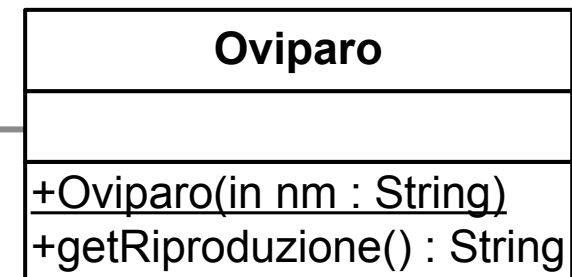
```
public abstract class Animale
{
    private String nome;

    public Animale(String nm)
    {
        nome=nm;
    }
    public String getNome()
    {
        return "Nome: "+nome;
    }
    public abstract String getMovimento();
    public abstract String getRiproduzione();
    public abstract String getCopertura();
    public String toString()
    {
        return getNome()+
            "Caratteristiche: "+getMovimento()+
            ", "+getRiproduzione()+", "+
            getCopertura();
    }
}
```

La classe Oviparo

- Oviparo è un'altra classe astratta e ci serve per fattorizzare il metodo **getRiproduzione()**

```
public abstract class Oviparo
    extends Animale
{
    public Oviparo(String nm)
    {
        super(nm);
    }
    public String getRiproduzione()
    {
        return "Oviparo";
    }
}
```



Overriding

Le classi Pesce ed Uccello

- Definiamo quindi le due classi concrete Pesce e Uccello
- Specifichiamo nelle classi (non astratte) i metodi non ancora implementati:

```
public class Pesce
    extends Oviparo
{
    public Pesce(String nm)
    { super(nm); }
    public String getMovimento()
    { return "Nuota"; }
    public String getCopertura()
    { return "Squame"; }
}
```

Pesce
+Pesce(in nm : String) +getMovimento() : String +getCopertura() : String

Uccello
+Uccello(in nm : String) +getMovimento() : String +getCopertura() : String

```
public class Uccello
    extends Oviparo
{
    public Uccello(String nm)
    { super(nm); }
    public String getMovimento()
    { return "Volare"; }
    public String getCopertura()
    { return "Piume"; }
}
```

La classe Mammifero

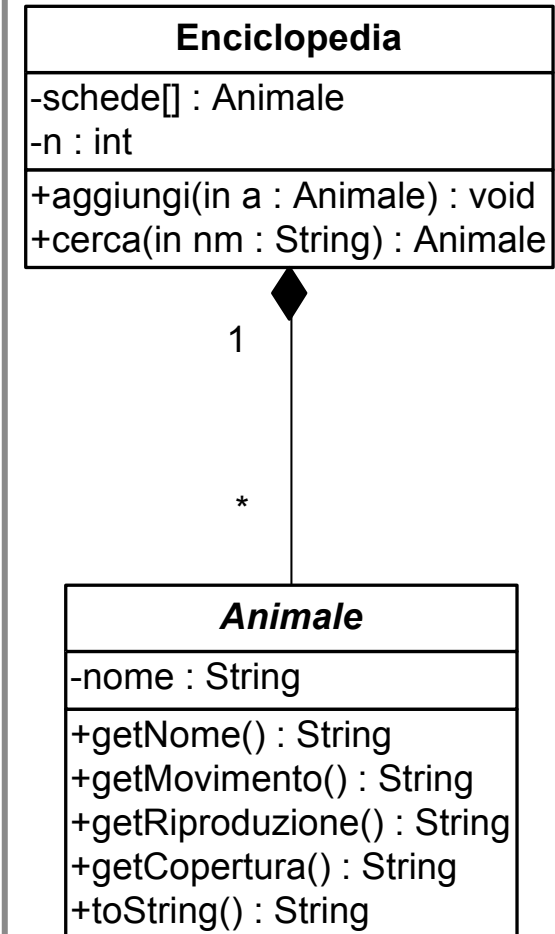
- Definiamo infine la classe concreta Mammifero che discende direttamente da Animale

```
public class Mammifero extends Animale
{
    public Mammifero(String nm)
    { super(nm); }
    public String getMovimento()
    { return "Cammina"; }
    public String getRiproduzione()
    { return "Viviparo"; }
    public String getCopertura()
    { return "Peli"; }
}
```

L'enciclopedia

- Enciclopedia sarà quindi una classe che contiene una collezione di schede:

```
public class Enciclopedia
{
    private Animale[] schede;
    private int n;
    public Enciclopedia(int max)
    { n = 0; schede = new Animale[max]; }
    public void aggiungi(Animale a)
    { schede[n] = a; n++; }
    public Animale cerca(String nm)
    {
        for (int i=0; i<n; i++)
            if (nm.equals(schede[i].getNome()))
                return schede[i];
        return null;
    }
}
```



La classe principale

```
import java.io.*;
public class EsempioEnciclopedia
{
    public static void main(String[] args) {
        Enciclopedia e = new Enciclopedia(50);
        e.aggiungi(new Mammifero("Cavallo"));
        e.aggiungi(new Mammifero("Cane"));
        e.aggiungi(new Pesce("Tonno"));
        e.aggiungi(new Uccello("Rondine"));

        Animale a;
        a = e.cerca("Tonno");
        if (a != null) System.out.println(a);
        else System.out.println("Non trovato")
    }
}
```

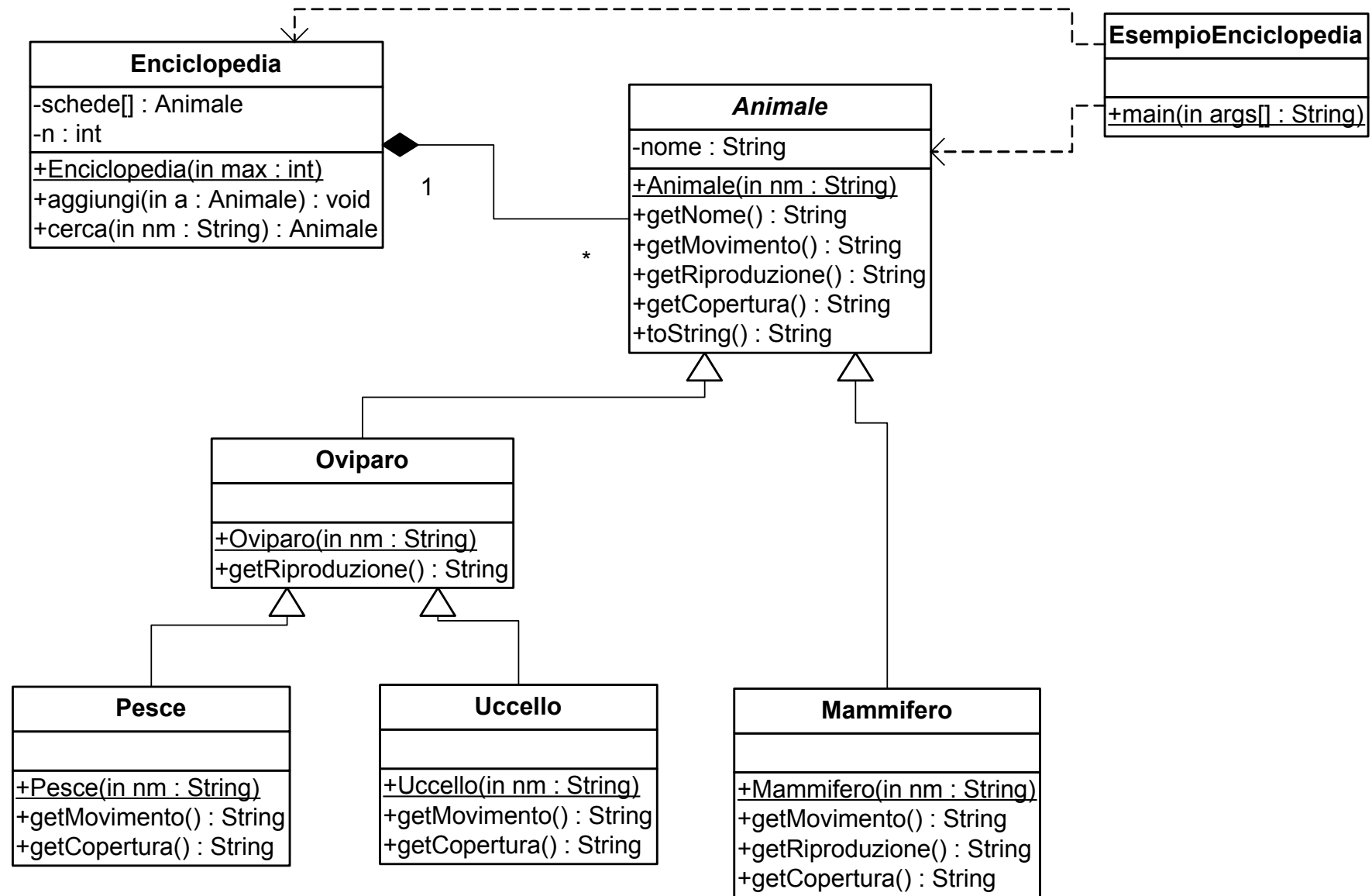
**Grazie al subtyping
possiamo trattare
indifferentemente i
discendenti di Animale**

La classe principale

```
import java.io.*;
public class EsempioEnciclopedia
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader kbd = new BufferedReader(new
            InputStreamReader(System.in));
        Enciclopedia e = new Enciclopedia(50);
        e.aggiungi(new Mammifero("Cavallo"));
        e.aggiungi(new Mammifero("Cane"));
        e.aggiungi(new Pesce("Tonno"));
        e.aggiungi(new Uccello("Rondine")); ...
        String st = ""; Animale a;
        while (st != "*")
        { System.out.println("Nome di animale o * per uscire");
          st = kbd.readLine(); a = e.cerca(st);
          if (a != null) System.out.println(a);
          else System.out.println("Non trovato");
        }
    }
}
```

**Grazie al subtyping
possiamo trattare
indifferentemente i
discendenti di Animale**

Il diagramma delle classi completo



La vita di solito è più complicata...

- La soluzione che abbiamo adottato va bene solo per un esempio semplificato come quello preso in considerazione
- Lascia diverse cose in sospeso:
 - Come gestiamo i Cetacei che, pur essendo mammiferi, nuotano e non hanno peli?
 - Se vogliamo gestire anche il tipo di respirazione (branchie, polmoni) la scelta di far discendere pesci e uccelli dalla classe Oviparo non è più vantaggiosa
- Una soluzione ben costruita al problema generale è più complessa e usa tecniche diverse da quella qui esposta
- **Non bisogna confondere gli esempi che servono ad illustrare un aspetto di un linguaggio o di un modello con le soluzioni di progettazione che devono essere adottate in situazioni reali**