INTRODUZIONE OOP
EREDITARIERÀ

“La programmazione OOP è caratterizzata da una proprietà detta ereditarietà, che consiste nel riutilizzare software preesistente, apportando le sole modifiche necessarie al problema in esame.”

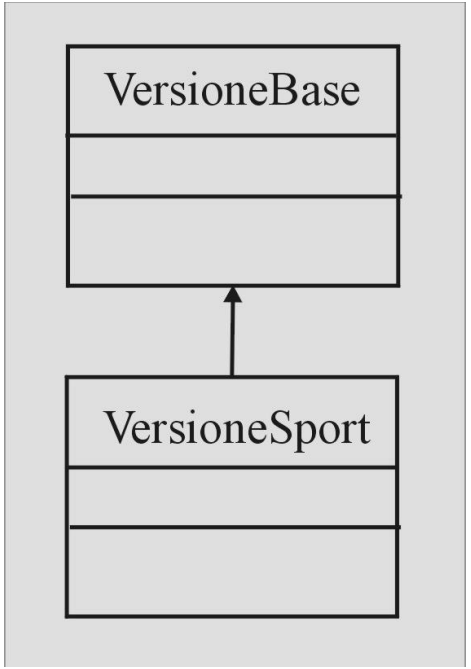
EREDITARIETÀ

Immaginiamo un modello di vettura. Essa viene prodotta in “versione base” ed in “versione sport”.
Ciò significa che il “modello sport” possiede le stesse caratteristiche del “modello base”, ma in più presenta altri requisiti **specifici**.
In altra parole, il “modello sport” **deriva** dal “modello base”, ossia ne **eredita** le proprietà e le funzionalità.

EREDITARIETÀ

La classe *VersioneSport* si dice **classe derivata** di *VersioneBase* , mentre *VersioneBase* si dice **classe base** di *VersioneSport* .
La **riusabilità** del codice comporta:

- diminuzione dei **tempi di sviluppo**
- possibilità di **espansione** del codice esistente
- possibilità di **ridefinizione** del codice esistente
- **indipendenza** dal codice sorgente usato nella classe base
- **testing** del solo codice espanso



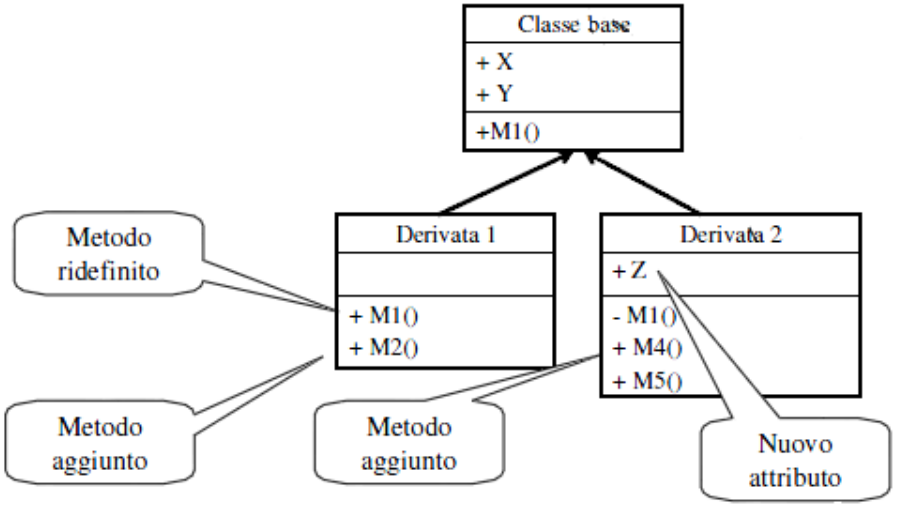
EREDITARIETÀ

Una classe derivata, in particolare:

- **eredita** tutti e soli i **membri pubblici** della classe base
- può contenere **espansioni dei membri** della classe base (nuovi attributi e nuovi metodi) che si aggiungono a quelli della classe base
- può richiedere la **ridefinizione** di metodi (**overriding**) ossia fornire una nuova implementazione di uno o più metodi ereditati dalla classe base

EREDITARIETÀ

La derivazione, da una classe base, di classi derivate per risolvere un problema, si rappresenta mediante l'**UML**, con una struttura detta **gerarchia delle classi / CLASS DIAGRAM**



EREDITARIETÀ

La relazione di ereditarietà fa sì che anche i *metodi di default* (costruttori) seguano un *preciso ordine di esecuzione* quando sono istanziati nelle classi derivate.
In particolare:
il costruttore di una classe derivata:

- chiama *per prima cosa il costruttore della sua classe base* per inizializzare gli attributi e poi il proprio;
- se è omesso, *chiama direttamente il costruttore di default della classe base*

EREDITARIETÀ - ESEMPIO

Da una classe base è possibile costruire altre classi dette derivate
Le altre classi assumono la forma della classe base, ma possono essere arricchite da altri elementi
Le classi derivate possono dichiarare nuovi attributi e metodi

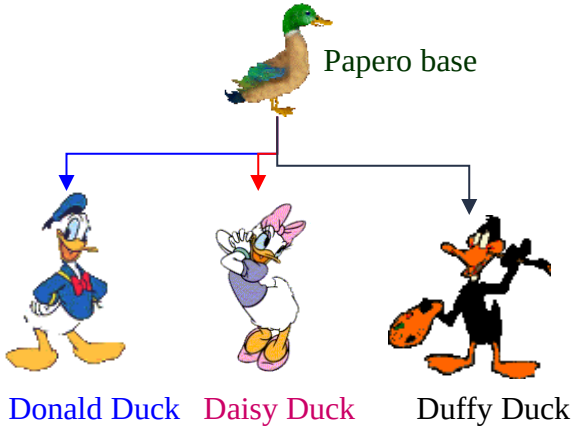
```
class Genitore {
 //attributi
 //metodi
}

class Figlio : Genitore {
 //altri attributi
 //altri metodi
}

Genitore G=new Genitore();
Figlio F=new Figlio();
```

EREDITARIETÀ

Gli oggetti possono essere costruiti a partire da altri oggetti

Formano una gerarchia

padre → figli



CLASSI BASE E CLASSI DERIVATE

Ogni classe derivata è un oggetto della propria classe base
L’ereditarietà forma una gerarchia rappresentabile come un albero
Per esprimere che la classe one è derivata dalla classe two si scrive

```
class one : two
```

CLASSI BASE E CLASSI DERIVATE

NB: **i costruttori non vengono ereditati**
Questo significa che occorre dotare ogni classe derivata dei propri costruttori.
I problemi sorgono quando occorre accedere ai dati(membri) della classe base.

ESEMPIO RISOLTO

Costruire la classe Point, definita da 2 coordinate, aggiungendo le proprietà opportune e l'override del metodo ToString(). Aggiungere poi la classe Circle, come derivata da Point, con in aggiunta il dato raggio. Scrivere le proprietà necessarie, l'override del metodo ToString(), oltre ai metodi Perimetro(), Diametro() e Area(). Scrivere un'applicazione client per testare il codice prodotto.

PROPOSTA SOLUZIONE

```
public class Point
{
    private int x;
    private int y;
```

Definizione classe Point

Definizione classe Circle

```
public class Circle : Point
{
    private double radius;
    public Circle()
    {}
    public Circle(int a,int b,double r):base(a,b)
    {
        this.radius = r;
    }
```

```
public override string ToString()
{
    return String.Format("Centre = [{0}]   Radius = {1}",
      base.ToString(),this.radius);
}
```

PROSEGUE ESERCIZIO

Partendo dalle classi definite prima, costruire la classe Cylinder, derivata da Circle, con in aggiunta il dato height. Scrivere le proprietà necessarie, l'override del metodo ToString(), l'override del metodo Area() (perché già esistente nella classe Circle, **COSA BISOGNA FARE?**) e il metodo Volume()
Scrivere un'applicazione client per testare il codice prodotto.

PERCHÈ?

Animale A = new Animale();
Mammifero M = new Mammifero();

A = M; //OK
OK poiché un Mammifero è anche un Animale

M = A; //NO!
NO poiché un Animale potrebbe NON essere un Mammifero